www.sapientia.uji.es | 18



Bases de datos

Mercedes Marqués

Bases de datos

Mercedes Marqués



DEPARTAMENT D'ENGINYERIA I CIÈNCIA DELS COMPUTADORS

■ Codi d'assignatura IG18

Edita: Publicacions de la Universitat Jaume I. Servei de Comunicació i Publicacions
Campus del Riu Sec. Edifici Rectorat i Serveis Centrals. 12071 Castelló de la Plana
http://www.tenda.uji.es e-mail: publicacions@uji.es

Col·lecció Sapientia, 18
Primera edició, 2011
www.sapientia.uji.es

ISBN: 978-84-693-0146-3



Índice general

Prefacio

Este texto se ha elaborado para dar soporte a un curso sobre *Bases de Datos* orientado a las *Ingenierías Informáticas*.

Los cuatro primeros capítulos realizan un estudio del modelo relacional: la estructura de datos, las reglas para mantener la integridad de la base de datos y los lenguajes relacionales, que se utilizan para manipular las bases de datos. Dentro de los lenguajes relacionales se hace una presentación exahustiva del lenguaje SQL, que es el lenguaje estándar de acceso a las bases de datos relacionales.

Los cuatro capítulos que vienen después plantean una metodología de diseño de bases de datos relacionales, comenzando por el diseño conceptual mediante el modelo entidad-relación. La siguiente etapa del diseño se aborda estableciendo una serie de reglas para obtener el esquema lógico de la base de datos, y la tercera y última etapa trata del diseño físico en SQL, al que se hace una introducción en el último capítulo de este texto.

Un estudio más profundo del diseño físico de bases de datos, así como el estudio de la funcionalidad de los sistemas de gestión de bases de datos, son temas que se deben incluir en un curso más avanzado sobre la materia.

Al comienzo de cada capítulo se incluye un apartado titulado *Introducción y objetivos* en el que se motiva el estudio del tema y se plantean los objetivos de aprendizaje que debe conseguir el estudiante. El texto incluye ejemplos y ejercicios resueltos, para ayudar a la comprensión de los contenidos. Este material se complementa con actividades a realizar por el estudiante, que serán publicadas en un entorno virtual de aprendizaje.

Aunque existe una amplia bibliografía sobre bases de datos, al final del texto se incluye sólo una breve selección de aquellos textos que han tenido más relevancia para la autora de estos apuntes.

Capítulo 1

Conceptos de bases de datos

Introducción y objetivos

El inicio de un curso sobre bases de datos debe ser, sin duda, la definición de *base de datos* y la presentación de los sistemas de gestión de bases de datos (el *software* que facilita la creación y manipulación de las mismas por parte del personal informático). Algunos de estos sistemas, ampliamente utilizados, son PostgreSQL, MySQL y Oracle.

Ya que este texto está dirigido a estudiantado de las ingenierías informáticas, es interesante conocer qué papeles puede desempeñar el personal informático en el entorno de una base de datos. Éstas han tenido sus predecesores en los sistemas de ficheros y tienen por delante un amplio horizonte, por lo que antes de comenzar su estudio resulta conveniente ubicarse en el tiempo haciendo un recorrido por su evolución histórica. El capítulo termina con una exposición sobre las ventajas y desventajas que las bases de datos conllevan.

Al finalizar este capítulo, el estudiantado debe ser capaz de:

- Definir qué es una base de datos y qué es un sistema de gestión de bases de datos.
- Reconocer los subsistemas que forman parte de un sistema de gestión de bases de datos.
- Enumerar las personas que aparecen en el entorno de una base de datos y sus tareas.
- Asociar los distintos tipos de sistemas de gestión de bases de datos a las generaciones a las que pertenecen.
- Enumerar las ventajas y desventajas de los sistemas de bases de datos y asociarlas al motivo por el que se producen: la integración de datos o el sistema de gestión de la base de datos.

1.1. Base de datos

Una *base de datos* es un conjunto de datos almacenados en memoria externa que están organizados mediante una estructura de datos. Cada base de datos ha sido diseñada para satisfacer los requisitos de información de una empresa u otro tipo de organización, como por ejemplo, una universidad o un hospital.

Antes de existir las bases de datos se trabajaba con *sistemas de ficheros*. Los sistemas de ficheros surgieron al informatizar el manejo de los archivadores manuales para proporcionar un acceso más eficiente a los datos almacenados en los mismos. Un sistema de ficheros sigue un modelo descentralizado, en el que cada departamento de la empresa almacena y gestiona sus propios datos mediante una serie de programas de aplicación escritos especialmente para él. Estos programas son totalmente independientes entre un departamento y otro, y se utilizan para introducir datos, mantener los ficheros y generar los informes que cada departamento necesita. Es importante destacar que en los sistemas de ficheros, tanto la estructura física de los ficheros de datos como la de sus registros, están definidas dentro de los programas de aplicación.

Cuando en una empresa se trabaja con un sistema de ficheros, los departamentos no comparten información ni aplicaciones, por lo que los datos comunes deben estar duplicados en cada uno de ellos. Esto puede originar inconsistencias en los datos. Se produce una inconsistencia cuando copias de los mismos datos no coinciden: dos copias del domicilio de un cliente pueden no coincidir si sólo uno de los departamentos que lo almacenan ha sido informado de que el domicilio ha cambiado.

Otro inconveniente que plantean los sistemas de ficheros es que cuando los datos se separan en distintos ficheros, es más complicado acceder a ellos, ya que el programador de aplicaciones debe sincronizar el procesamiento de los distintos ficheros implicados para garantizar que se extraen los datos correctos. Además, ya que la estructura física de los datos se encuentra especificada en los programas de aplicación, cualquier cambio en dicha estructura es difícil de realizar. El programador debe identificar todos los programas afectados por el cambio, modificarlos y volverlos a probar, lo que cuesta mucho tiempo y está sujeto a que se produzcan errores. A este problema, tan característico de los sistemas de ficheros, se le denomina también falta de *independencia de datos lógica-física*.

Una base de datos se puede percibir como un gran almacén de datos que se define y se crea una sola vez, y que se utiliza al mismo tiempo por distintos usuarios. En una base de datos todos los datos se integran con una mínima cantidad de duplicidad. De este modo, la base de datos no pertenece a un solo departamento sino que se comparte por toda la organización. Además, la base de datos no sólo contiene los datos de la organización, también almacena una descripción de dichos datos. Esta descripción es lo que se denomina *metadatos*, se almacena en el *diccionario de datos* o *catálogo* y es lo que permite que exista independencia de datos lógica-física.

1.2. Sistema de gestión de bases de datos

El *sistema de gestión de la base de datos* (en adelante SGBD) es una aplicación que permite a los usuarios definir, crear y mantener la base de datos, además de proporcionar un acceso controlado a la misma. Se denomina *sistema de bases de datos* al conjunto formado por la base de datos, el SGBD y los programas de aplicación que dan servicio a la empresa u organización.

El modelo seguido con los sistemas de bases de datos es muy similar al modelo que se sigue en la actualidad para el desarrollo de programas con lenguajes orientados a objetos, en donde se da una implementación interna de un objeto y una especificación externa separada. Los usuarios del objeto sólo ven la especificación externa y no se deben preocupar de cómo se implementa internamente el objeto. Una ventaja de este modelo, conocido como abstracción de datos, es que se puede cambiar la implementación interna de un objeto sin afectar a sus usuarios ya que la especificación externa no se ve alterada. Del mismo modo, los sistemas de bases de datos separan la definición de la estructura física de los datos de su estructura lógica, y almacenan esta definición en la base de datos. Todo esto es gracias a la existencia del SGBD, que se sitúa entre la base de datos y los programas de aplicación.

Generalmente, un SGBD proporciona los servicios que se citan a continuación:

- El SGBD permite la definición de la base de datos mediante un *lenguaje de definición de datos*. Este lenguaje permite especificar la estructura y el tipo de los datos, así como las restricciones sobre los datos.
- El SGBD permite la inserción, actualización, eliminación y consulta de datos mediante un *lenguaje de manejo de datos*. El hecho de disponer de un lenguaje para realizar consultas reduce el problema de los sistemas de ficheros, en los que el usuario tiene que trabajar con un conjunto fijo de consultas, o bien, dispone de un gran número de programas de aplicación costosos de gestionar. Hay dos tipos de lenguajes de manejo de datos: los *procedurales* y los *no procedurales*. Estos dos tipos se distinguen por el modo en que acceden a los datos. Los lenguajes procedurales manipulan la base de datos registro a registro, mientras que los no procedurales operan sobre conjuntos de registros. En los lenguajes procedurales se especifica qué operaciones se debe realizar para obtener los datos resultado, mientras que en los lenguajes no procedurales se especifica qué datos deben obtenerse sin decir cómo hacerlo. El lenguaje no procedural más utilizado es el SQL (*Structured Query Language*) que, de hecho, es un estándar y es el lenguaje de los SGBD relacionales.
- El SGBD proporciona un acceso controlado a la base de datos mediante:
  - Un sistema de seguridad, de modo que los usuarios no autorizados no puedan acceder a la base de datos.

- Un sistema de integridad que mantiene la integridad y la consistencia de los datos.
- Un sistema de control de concurrencia que permite el acceso compartido a la base de datos.
- Un sistema de control de recuperación que restablece la base de datos después de que se produzca un fallo del *hardware* o del *software*.
- Un diccionario de datos o catálogo, accesible por el usuario, que contiene la descripción de los datos de la base de datos.

A diferencia de los sistemas de ficheros, en los que los programas de aplicación trabajan directamente sobre los ficheros de datos, el SGBD se ocupa de la estructura física de los datos y de su almacenamiento. Con esta funcionalidad, el SGBD se convierte en una herramienta de gran utilidad. Sin embargo, desde el punto de vista del usuario, se podría discutir que los SGBD han hecho las cosas más complicadas, ya que ahora los usuarios ven más datos de los que realmente quieren o necesitan, puesto que ven la base de datos completa. Conscientes de este problema, los SGBD proporcionan un mecanismo de *vistas* que permite que cada usuario tenga su propia vista o visión de la base de datos. El lenguaje de definición de datos permite definir vistas como subconjuntos de la base de datos.

Todos los SGBD no presentan la misma funcionalidad, depende de cada producto. En general, los grandes SGBD multiusuario ofrecen todas las funciones que se acaban de citar e incluso más. Los sistemas modernos son conjuntos de programas extremadamente complejos y sofisticados, con millones de líneas de código y con una documentación consistente en varios volúmenes. Lo que se pretende es proporcionar un sistema que permita gestionar cualquier tipo de requisitos y que tenga un 100 % de fiabilidad ante cualquier tipo de fallo. Los SGBD están en continua evolución, tratando de satisfacer los requisitos de todo tipo de usuarios. Por ejemplo, muchas aplicaciones de hoy en día necesitan almacenar imágenes, vídeo, sonido, etc. Para satisfacer a este mercado, los SGBD deben evolucionar. Conforme vaya pasando el tiempo, irán surgiendo nuevos requisitos, por lo que los SGBD nunca permanecerán estáticos.

1.3. Personas en el entorno de las bases de datos

Hay cuatro grupos de personas que intervienen en el entorno de una base de datos: el administrador de la base de datos, los diseñadores de la base de datos, los programadores de aplicaciones y los usuarios.

El *administrador de la base de datos* se encarga de la implementación física de la base de datos: escoge los tipos de los ficheros de datos y de los índices que deben crearse, determina dónde deben ubicarse ficheros e índices y, en general,

toma las decisiones relativas al almacenamiento físico en función de las posibilidades que le ofrezca el SGBD con el que trabaje. Además, el administrador de la base de datos se encarga de establecer la política de seguridad y del acceso concurrente. También se debe preocupar de que el sistema se encuentre siempre operativo y procurar que los usuarios y las aplicaciones obtengan buenas prestaciones. El administrador debe conocer muy bien el SGBD con el que trabaja, así como el equipo informático sobre el que esté funcionando.

Los *diseñadores de la base de datos* realizan el diseño de la base de datos, debiendo identificar los datos, las relaciones entre ellos y las restricciones sobre los datos y sobre sus relaciones. El diseñador de la base de datos debe tener un profundo conocimiento de los datos de la empresa y también debe conocer sus *reglas de negocio*. Las reglas de negocio describen las características principales sobre el comportamiento de los datos tal y como las ve la empresa. Para obtener un buen resultado, el diseñador de la base de datos debe implicar en el proceso a todos los usuarios de la base de datos, tan pronto como sea posible.

Una vez se ha diseñado e implementado la base de datos, los *programadores de aplicaciones* se encargan de implementar los programas de aplicación que servirán a los usuarios finales. Estos programas de aplicación son los que permiten consultar datos, insertarlos, actualizarlos y eliminarlos. Estos programas se escriben mediante lenguajes de tercera generación o de cuarta generación.

Los *usuarios finales* son los clientes de la base de datos: la base de datos ha sido diseñada e implementada, y está siendo mantenida, para satisfacer sus requisitos en la gestión de su información.

1.4. Historia de los sistemas de bases de datos

Los predecesores de los sistemas de bases de datos fueron los sistemas de ficheros. Un sistema de ficheros está formado por un conjunto de ficheros de datos y los programas de aplicación que permiten a los usuarios finales trabajar sobre los mismos. No hay un momento concreto en el que los sistemas de ficheros hayan cesado y hayan dado comienzo los sistemas de bases de datos. De hecho, todavía existen sistemas de ficheros en uso.

Se dice que los sistemas de bases de datos tienen sus raíces en el proyecto estadounidense de mandar al hombre a la luna en los años sesenta, el proyecto Apolo. En aquella época, no había ningún sistema que permitiera gestionar la inmensa cantidad de información que requería el proyecto. La primera empresa encargada del proyecto, NAA (*North American Aviation*), desarrolló una aplicación denominada GUAM (*General Update Access Method*) que estaba basada en el concepto de que varias piezas pequeñas se unen para formar una pieza más grande, y así sucesivamente hasta que el producto final está ensamblado. Esta estructura, que tiene la forma de un árbol, es lo que se denomina una *estructura jerárquica*. A mediados de los sesenta, IBM se unió a NAA para desarrollar GUAM en lo que después fue IMS (*Information Management*

System). El motivo por el cual IBM restringió IMS al manejo de jerarquías de registros fue el de permitir el uso de dispositivos de almacenamiento serie, más exactamente las cintas magnéticas, ya que era un requisito del mercado por aquella época.

A mitad de los sesenta, General Electric desarrolló IDS (*Integrated Data Store*). Este trabajo fue dirigido por uno de los pioneros en los sistemas de bases de datos, Charles Bachmann. IDS era un nuevo tipo de sistema de bases de datos conocido como *sistema de red*, que produjo un gran efecto sobre los sistemas de información de aquella generación. El sistema de red se desarrolló, en parte, para satisfacer la necesidad de representar relaciones entre datos más complejas que las que se podían modelar con los sistemas jerárquicos y, en parte, para imponer un estándar de bases de datos. Para ayudar a establecer dicho estándar, el grupo CODASYL (*Conference on Data Systems Languages*), formado por representantes del gobierno de EEUU y representantes del mundo empresarial, fundaron un grupo denominado DBTG (*Data Base Task Group*), cuyo objetivo era definir unas especificaciones estándar que permitieran la creación de bases de datos y el manejo de los datos. El DBTG presentó su informe final en 1971 y aunque éste no fue formalmente aceptado por ANSI (*American National Standards Institute*), muchos sistemas se desarrollaron siguiendo la propuesta del DBTG. Estos sistemas son los que se conocen como sistemas de red, sistemas CODASYL o DBTG.

Los sistemas jerárquico y de red constituyen la *primera generación* de los SGBD. Estos sistemas presentan algunos inconvenientes:

- Es necesario escribir complejos programas de aplicación para responder a cualquier tipo de consulta de datos, por simple que ésta sea.
- La independencia de datos es mínima.
- No tienen un fundamento teórico.

En 1970, Edgar Frank Codd de los laboratorios de investigación de IBM, escribió un artículo presentando el *modelo relacional*. En este artículo presentaba también los inconvenientes de los sistemas previos, el jerárquico y el de red. Pasó casi una década hasta que se desarrollaron los primeros sistemas relacionales. Uno de los primeros es System R, de IBM, que se desarrolló para probar la funcionalidad del modelo relacional, proporcionando una implementación de sus estructuras de datos y sus operaciones. Esto condujo a dos grandes desarrollos:

- El desarrollo de un lenguaje de consultas estructurado denominado SQL, que se ha convertido en el lenguaje estándar de los sistemas relacionales.
- La producción de varios SGBD relacionales durante los años ochenta, como DB2 y SLQ/DS, de IBM, y Oracle, de Oracle Corporation.

Hoy en día, existen cientos de SGBD relacionales, tanto para microordenadores como para sistemas multiusuario, aunque muchos no son completamente fieles al modelo relacional.

Los SGBD relacionales constituyen la *segunda generación* de los SGBD. Sin embargo, el modelo relacional también tiene sus debilidades, siendo una de ellas su limitada capacidad al modelar los datos. Se ha desarrollado mucha investigación desde entonces tratando de resolver este problema. En 1976, Peter Chen presentó el modelo entidad-relación, que es la técnica más utilizada en el diseño de bases de datos. En 1979, Codd intentó subsanar algunas de las deficiencias de su modelo relacional con una versión extendida denominada RM/T (1979) y más recientemente RM/V2 (1990). Los intentos de proporcionar un modelo de datos que represente al mundo real de un modo más fiel han dado lugar a los modelos de datos semánticos.

La evolución reciente de la tecnología de bases de datos viene marcada por una mayor solidez en las bases de datos orientadas a objetos, la extensión de las bases de datos relacionales y el procesamiento distribuido. Esta evolución representa la *tercera generación* de los SGBD.

Por su parte, los sistemas de gestión de bases de datos relacionales han ido evolucionando estos últimos años para soportar objetos y reglas, y para ampliar el lenguaje SQL y hacerlo más extensible y computacionalmente completo, dando lugar a lo que se conoce como sistemas objeto-relacionales.

Durante la última década, el impacto de los avances en la tecnología de las comunicaciones ha sido muy importante. Esto ha contribuido a que en las empresas se haya producido una mayor distribución de la gestión automática de la información, en contraste con la filosofía centralizadora predominante en la tecnología inicial de bases de datos. Las bases de datos distribuidas posibilitan el procesamiento de datos pertenecientes a distintas bases de datos conectadas entre sí. El emplazamiento lógico de cada una de las bases de datos se denomina nodo, conteniendo cada uno su sistema de gestión de bases de datos, junto con las utilidades y facilidades propias del soporte distribuido. Los nodos, por lo general, están ubicados en emplazamientos físicos distantes geográficamente, y se encuentran conectados por una red de comunicación de datos.

Por otra parte, los sistemas de bases de datos activas han sido propuestos como otro paradigma de gestión de datos que satisface las necesidades de aquellas aplicaciones que requieren una respuesta puntual ante situaciones críticas. Como ejemplos se puede citar el control del tráfico aéreo o las aplicaciones de control de plantas industriales. Este paradigma también puede ser utilizado para soportar varias de las funciones del propio sistema de gestión de bases de datos, como son: el control de accesos, el control de la integridad, el mantenimiento de vistas o el mantenimiento de atributos derivados. El factor común en todas estas aplicaciones es la necesidad de responder a sucesos, tanto externos como internos al propio sistema. A diferencia de los sistemas pasivos, un sistema de gestión de bases de datos activas responde automáticamente

ante determinadas circunstancias descritas por el diseñador. La mayoría de los sistemas de gestión de bases de datos comerciales incorporan la posibilidad de definir reglas, por lo que son, en cierto modo, sistemas activos.

Las investigaciones sobre la relación entre la teoría de las bases de datos y la lógica se remontan a finales de la década de los setenta. Estas investigaciones han dado lugar a las bases de datos deductivas, que permiten derivar nuevas informaciones a partir de las introducidas explícitamente por el usuario. Esta función deductiva se realiza mediante la adecuada explotación de ciertas reglas de conocimiento relativas al dominio de la aplicación, utilizando para ello técnicas de programación lógica y de inteligencia artificial.

Los sistemas de múltiples bases de datos permiten realizar operaciones que implican a varios sistemas de bases de datos, cada uno de los cuales puede ser centralizado o distribuido. Cada sistema de bases de datos que participa es denominado *componente*. Si todos los sistemas de gestión de bases de datos de los diferentes componentes son iguales, el sistema de múltiples bases de datos es homogéneo; en caso contrario, es heterogéneo. Un sistema de múltiples bases de datos es un sistema federado de bases de datos si permite una doble gestión: una de carácter global, realizada por el sistema de gestión de bases de datos federadas y otra en modo autónomo e independiente del sistema federado, realizada por parte de los sistemas componentes.

La influencia de la Web lo abarca todo. En su desarrollo se han ignorado las técnicas de bases de datos, por lo que se han repetido los errores cometidos en las primeras generaciones de los sistemas de gestión de bases de datos. La Web se puede ver como una nueva interfaz de acceso a bases de datos, y muchos sistemas de gestión de bases de datos ya proporcionan almacenamiento y acceso a datos a través de XML. Pero la Web puede también ser considerada como una inmensa base de datos, siendo éste un tema de investigación en pleno auge.

Por otra parte, los grandes almacenes de datos (*data warehouses*) ya han demostrado que si son implementados convenientemente, pueden ser de gran ayuda en la toma de decisiones y en el procesamiento analítico en tiempo real OLAP (*On-Line Analytical Processing*). Los datos son extraídos periódicamente de otras fuentes y son integrados en el almacén. Estos datos, relevantes para la empresa, son no-volátiles y se agrupan según diversas granularidades en el tiempo y en otras dimensiones. En la actualidad, existe una gran competencia entre las extensiones de los sistemas de gestión de bases de datos comerciales para incorporar las características de este tipo de sistemas, y la creación de productos específicos.

La explotación de datos (*data mining* o *knowledge discovery in databases*) trata de descubrir conocimientos útiles y previamente no conocidos a partir de grandes volúmenes de datos, por lo que no sólo integra técnicas de bases de datos, sino también de estadística y de inteligencia artificial. Las investigaciones se han plasmado rápidamente en productos comerciales, con un desarrollo reciente bastante importante.

Existen también muchos trabajos de investigación en temas tales como las bases de datos temporales y las bases de datos multimedia. Las bases de datos temporales intentan, en primer lugar, definir un modelo de datos que capture la semántica del tiempo en el mundo real, y, en segundo lugar, realizar una implementación eficiente de tal modelo. Los recientes avances en el almacenamiento de distintos tipos de información, como voz, imágenes o sonido, han tenido su influencia en las bases de datos, dando lugar a las bases de datos multimedia.

La rápida evolución que la tecnología de bases de datos ha experimentado en la última década, así como la variedad de nuevos caminos abiertos, han conducido a investigadores y asociaciones interesadas, a reflexionar sobre el futuro de esta tecnología. Estas reflexiones quedan recogidas en numerosos debates y manifiestos que intentan poner orden en un campo en continua expansión.

1.5. Ventajas e inconvenientes de los sistemas de bases de datos

Los sistemas de bases de datos presentan numerosas ventajas gracias, fundamentalmente, a la integración de datos y a la interfaz común que proporciona el SGBD. Estas ventajas se describen a continuación.

- *Control sobre la redundancia de datos.* Los sistemas de ficheros almacenan varias copias de los mismos datos en ficheros distintos. Esto hace que se desperdicie espacio de almacenamiento, además de provocar faltas de consistencia de datos (copias que no coinciden). En los sistemas de bases de datos todos estos ficheros están integrados, por lo que no se almacenan varias copias de los mismos datos. Sin embargo, en una base de datos no se puede eliminar la redundancia completamente, ya que en ocasiones es necesaria para modelar las relaciones entre los datos, o bien es necesaria para mejorar las prestaciones.

- *Control sobre la consistencia de datos.* Eliminando o controlando las redundancias de datos se reduce en gran medida el riesgo de que haya inconsistencias. Si un dato está almacenado una sola vez, cualquier actualización se debe realizar sólo una vez, y está disponible para todos los usuarios inmediatamente. Si un dato está duplicado y el sistema conoce esta redundancia, el propio sistema puede encargarse de garantizar que todas las copias se mantengan consistentes. Desgraciadamente, no todos los SGBD de hoy en día se encargan de mantener automáticamente la consistencia.

- *Compartición de datos.* En los sistemas de ficheros, los ficheros pertenecen a los departamentos que los utilizan, pero en los sistemas de bases de

datos, la base de datos pertenece a la empresa y puede ser compartida por todos los usuarios que estén autorizados. Además, las nuevas aplicaciones que se vayan creando pueden utilizar los datos de la base de datos existente.

- *Mantenimiento de estándares.* Gracias a la integración es más fácil respetar los estándares necesarios, tanto los establecidos a nivel de la empresa como los nacionales e internacionales. Estos estándares pueden establecerse sobre el formato de los datos para facilitar su intercambio; pueden ser estándares de documentación, procedimientos de actualización y también reglas de acceso.

- *Mejora en la integridad de datos.* La integridad de la base de datos se refiere a la validez de los datos almacenados. Normalmente, la integridad se expresa mediante restricciones o reglas que no se pueden violar. Estas restricciones se pueden aplicar tanto a los datos, como a sus relaciones, y es el SGBD quien se encargará de mantenerlas.

- *Mejora en la seguridad.* La seguridad de la base de datos consiste la protección de la base de datos frente a usuarios no autorizados. Sin unas buenas medidas de seguridad, la integración de datos en los sistemas de bases de datos hace que éstos sean más vulnerables que en los sistemas de ficheros. Sin embargo, los SGBD permiten mantener la seguridad mediante el establecimiento de claves para identificar al personal autorizado a utilizar la base de datos. Las autorizaciones se pueden realizar a nivel de operaciones, de modo que un usuario puede estar autorizado a consultar ciertos datos pero no a actualizarlos, por ejemplo.

- *Mejora en la accesibilidad a los datos.* Muchos SGBD proporcionan lenguajes de consulta o generadores de informes que permiten al usuario hacer cualquier tipo de consulta sobre los datos, sin que sea necesario que un programador escriba una aplicación que realice tal tarea.

- *Mejora en la productividad.* El SGBD proporciona muchas de las funciones estándar que el programador necesita escribir en un sistema de ficheros. A nivel básico, el SGBD proporciona todas las rutinas de manejo de ficheros típicas de los programas de aplicación. El hecho de disponer de estas funciones permite al programador centrarse mejor en la función específica requerida por los usuarios, sin tener que preocuparse de los detalles de implementación de bajo nivel. Muchos SGBD también proporcionan un entorno de cuarta generación consistente en un conjunto de herramientas que simplifican, en gran medida, el desarrollo de las aplicaciones que acceden a la base de datos. Gracias a estas herramientas, el programador puede ofrecer una mayor productividad en un tiempo menor.

- *Mejora en el mantenimiento* gracias a la independencia de datos. En los sistemas de ficheros, las descripciones de los datos se encuentran inmersas en los programas de aplicación que los manejan. Esto hace que los programas sean dependientes de los datos, de modo que un cambio en su estructura, o un cambio en el modo en que se almacena en disco, requiere cambios importantes en los programas cuyos datos se ven afectados. Sin embargo, los SGBD separan las descripciones de los datos de las aplicaciones. Esto es lo que se conoce como independencia de datos, gracias a la cual se simplifica el mantenimiento de las aplicaciones que acceden a la base de datos.

- *Aumento de la concurrencia.* En algunos sistemas de ficheros, si hay varios usuarios que pueden acceder simultáneamente a un mismo fichero, es posible que el acceso interfiera entre ellos de modo que se pierda información o, incluso, que se pierda la integridad. La mayoría de los SGBD gestionan el acceso concurrente a la base de datos y pueden garantizar que no ocurran problemas de este tipo.

- *Mejora en los servicios de copias de seguridad y de recuperación ante fallos.* Muchos sistemas de ficheros dejan que sea el usuario quien proporcione las medidas necesarias para proteger los datos ante fallos en el sistema o en las aplicaciones. Los usuarios tienen que hacer copias de seguridad cada día, y si se produce algún fallo, utilizar estas copias para restaurarlos. En este caso, todo el trabajo realizado sobre los datos desde que se hizo la última copia de seguridad se pierde y se tiene que volver a realizar. Sin embargo, los SGBD actuales funcionan de modo que se minimiza la cantidad de trabajo perdido cuando se produce un fallo.

La integración de los datos y la existencia del SGBD también plantean ciertos inconvenientes, como los que se citan a continuación.

- *Alta complejidad.* Los SGBD son conjuntos de programas muy complejos con una gran funcionalidad. Es preciso comprender muy bien esta funcionalidad para poder sacar un buen partido de ellos.

- *Gran tamaño.* Los SGBD son programas complejos y muy extensos que requieren una gran cantidad de espacio en disco y de memoria para trabajar de forma eficiente.

- *Coste económico del SGBD.* El coste de un SGBD varía dependiendo del entorno y de la funcionalidad que ofrece. Por ejemplo, un SGBD para un ordenador personal puede costar 500 €, mientras que un SGBD para un sistema multiusuario que dé servicio a cientos de usuarios puede costar entre 10000 y 100000 €. Además, hay que pagar una cuota anual de mantenimiento que suele ser un porcentaje del precio del SGBD. En los últimos años han surgido SGBD libres (*open source*) que ofrecen una gran funcionalidad y muy buenas prestaciones.

- *Coste del equipamiento adicional.* Tanto el SGBD, como la propia base de datos, pueden hacer que sea necesario adquirir más espacio de almacenamiento. Además, para alcanzar las prestaciones deseadas, es posible que sea necesario adquirir una máquina más grande o una máquina que se dedique solamente al SGBD. Todo esto hará que la implantación de un sistema de bases de datos sea más cara.

- *Coste de la conversión.* En algunas ocasiones, el coste del SGBD y el coste del equipo informático que sea necesario adquirir para su buen funcionamiento es insignificante comparado al coste de convertir la aplicación actual en un sistema de bases de datos. Este coste incluye el coste de enseñar a la plantilla a utilizar estos sistemas y, probablemente, el coste del personal especializado para ayudar a realizar la conversión y poner en marcha el sistema. Este coste es una de las razones principales por las que algunas empresas y organizaciones se resisten a cambiar su sistema actual de ficheros por un sistema de bases de datos.

- *Prestaciones.* Un sistema de ficheros está escrito para una aplicación específica, por lo que sus prestaciones suelen ser muy buenas. Sin embargo, los SGBD están escritos para ser más generales y ser útiles en muchas aplicaciones, lo que puede hacer que algunas de ellas no sean tan rápidas como antes.

- *Vulnerable a los fallos.* El hecho de que todo esté centralizado en el SGBD hace que el sistema sea más vulnerable ante los fallos que puedan producirse.

Capítulo 2

Modelo relacional

Introducción y objetivos

En este capítulo se presentan los principios básicos del modelo relacional, que es el modelo de datos en el que se basan la mayoría de los SGBD en uso hoy en día. En primer lugar, se presenta la estructura de datos relacional y a continuación las reglas de integridad que deben cumplirse sobre la misma.

Al finalizar este capítulo, el estudiante debe ser capaz de:

- Definir qué es un modelo de datos y describir cómo se clasifican los modelos de datos.
- Definir los distintos modelos lógicos de bases de datos.
- Definir la estructura de datos relacional y todas sus partes.
- Enumerar las propiedades de las relaciones.
- Definir los tipos de relaciones.
- Definir superclave, clave candidata, clave primaria y clave ajena.
- Definir el concepto de nulo.
- Definir la regla de integridad de entidades y la regla de integridad referencial.
- Definir qué es una regla de negocio.
- Dar un ejemplo completo de una base de datos formada por, al menos, dos relaciones con claves ajenas.

2.1. Modelos de datos

Una de las características fundamentales de los sistemas de bases de datos es que proporcionan cierto nivel de abstracción de datos, al ocultar las características sobre el almacenamiento físico que la mayoría de usuarios no necesita conocer. Los *modelos de datos* son el instrumento principal para ofrecer dicha abstracción a través de su jerarquía de niveles. Un modelo de datos es un conjunto de conceptos que sirven para describir la estructura de una base de datos, es decir, los datos, las relaciones entre los datos y las restricciones que deben cumplirse sobre los datos. Los modelos de datos contienen también un conjunto de operaciones básicas para la realización de consultas (lecturas) y actualizaciones de datos. Además, los modelos de datos más modernos incluyen mecanismos para especificar acciones compensatorias o adicionales que se deben llevar a cabo ante las acciones habituales que se realizan sobre la base de datos.

Los modelos de datos se pueden clasificar dependiendo de los tipos de conceptos que ofrecen para describir la estructura de la base de datos, formando una jerarquía de niveles. Los modelos de datos de alto nivel, o *modelos conceptuales*, disponen de conceptos muy cercanos al modo en que la mayoría de los usuarios percibe los datos, mientras que los modelos de datos de bajo nivel, o *modelos físicos*, proporcionan conceptos que describen los detalles de cómo se almacenan los datos en el ordenador. Los conceptos de los modelos físicos están dirigidos al personal informático, no a los usuarios finales. Entre estos dos extremos se encuentran los *modelos lógicos*, cuyos conceptos pueden ser entendidos por los usuarios finales, aunque no están demasiado alejados de la forma en que los datos se organizan físicamente. Los modelos lógicos ocultan algunos detalles de cómo se almacenan los datos, pero pueden implementarse de manera directa en un SGBD.

Los modelos conceptuales utilizan conceptos como entidades, atributos y relaciones. Una *entidad* representa un objeto o concepto del mundo real como, por ejemplo, un cliente de una empresa o una de sus facturas. Un *atributo* representa alguna propiedad de interés de una entidad como, por ejemplo, el nombre o el domicilio del cliente. Una *relación* describe una interacción entre dos o más entidades, por ejemplo, la relación que hay entre un cliente y las facturas que se le han realizado.

Cada SGBD soporta un modelo lógico, siendo los más comunes el *relacional*, el de *red* y el *jerárquico*. Estos modelos representan los datos valiéndose de estructuras de registros, por lo que también se denominan *modelos orientados a registros*. Hay una familia más moderna de modelos lógicos, son los *modelos orientados a objetos*, que están más próximos a los modelos conceptuales. En el modelo relacional los datos se describen como un conjunto de tablas con referencias lógicas entre ellas, mientras que en los modelos jerárquico y de red, los datos se describen como conjuntos de registros con referencias físicas entre ellos (punteros).

Los modelos físicos describen cómo se almacenan los datos en el ordenador: el formato de los registros, la estructura de los ficheros (desordenados, ordenados, agrupados) y los métodos de acceso utilizados (índices, tablas de dispersión).

A la descripción de una base de datos mediante un modelo de datos se le denomina *esquema de la base de datos*. Este esquema se especifica durante el diseño, y no es de esperar que se modifique a menudo. Sin embargo, los datos que se almacenan en la base de datos pueden cambiar con mucha frecuencia: se insertan datos, se actualizan, se borran, etc. Los datos que la base de datos contiene en un determinado momento conforman el *estado de la base de datos*, o como también se denomina: una *ocurrencia de la base de datos*.

La distinción entre el esquema y el estado de la base de datos es muy importante. Cuando definimos una nueva base de datos, sólo especificamos su esquema al SGBD. En ese momento, el estado de la base de datos es el *estado vacío*, sin datos. Cuando se cargan datos por primera vez, la base datos pasa al *estado inicial*. De ahí en adelante, siempre que se realice una operación de actualización de la base de datos, se tendrá un nuevo estado. El SGBD se encarga, en parte, de garantizar que todos los estados de la base de datos sean estados válidos que satisfagan la estructura y las restricciones especificadas en el esquema. Por lo tanto, es muy importante que el esquema que se especifique al SGBD sea correcto y se debe tener gran cuidado al diseñarlo. El SGBD almacena el esquema en su *catálogo* o *diccionario de datos*, de modo que se pueda consultar siempre que sea necesario.

En 1970, el modo en que se veían las bases de datos cambió por completo cuando E. F. Codd introdujo el modelo relacional. En aquellos momentos, el enfoque existente para la estructura de las bases de datos utilizaba punteros físicos (direcciones de disco) para relacionar registros de distintos ficheros. Si, por ejemplo, se quería relacionar un registro A con un registro B, se debía añadir al registro A un campo conteniendo la dirección en disco (un puntero físico) del registro B. Codd demostró que estas bases de datos limitaban en gran medida los tipos de operaciones que los usuarios podían realizar sobre los datos. Además, estas bases de datos eran muy vulnerables a cambios en el entorno físico. Si se añadían los controladores de un nuevo disco al sistema y los datos se movían de una localización física a otra, se requería una conversión de los ficheros de datos. Estos sistemas se basaban en el modelo de red y el modelo jerárquico, los dos modelos lógicos que constituyeron la primera generación de los SGBD.

El modelo relacional representa la segunda generación de los SGBD. En él, todos los datos están estructurados a nivel lógico como tablas formadas por filas y columnas, aunque a nivel físico pueden tener una estructura completamente distinta. Un punto fuerte del modelo relacional es la sencillez de su estructura lógica. Pero detrás de esa simple estructura hay un fundamento teórico importante del que carecen los SGBD de la primera generación, lo que constituye otro punto a su favor.

Dada la popularidad del modelo relacional, muchos sistemas de la primera generación se han modificado para proporcionar una interfaz de usuario relacional, con independencia del modelo lógico que soportan (de red o jerárquico).

En los últimos años, se han propuesto algunas extensiones al modelo relacional para capturar mejor el significado de los datos, para disponer de los conceptos de la orientación a objetos y para disponer de capacidad deductiva.

El modelo relacional, como todo modelo de datos, tiene que ver con tres aspectos de los datos, que son los que se presentan en los siguientes apartados de este capítulo: qué características tiene la estructura de datos, cómo mantener la integridad de los datos y cómo realizar el manejo de los mismos.

2.2. Estructura de datos relacional

La estructura de datos del modelo relacional es la *relación*. En este apartado se presenta esta estructura de datos, sus propiedades, los tipos de relaciones y qué es una clave de una relación. Para facilitar la comprensión de las definiciones formales de todos estos conceptos, se dan antes unas definiciones informales que permiten asimilar dichos conceptos con otros que resulten familiares.

2.2.1. Relaciones

Definiciones informales

El modelo relacional se basa en el concepto matemático de *relación*, que gráficamente se representa mediante una tabla. Codd, que era un experto matemático, utilizó una terminología perteneciente a las matemáticas, en concreto de la teoría de conjuntos y de la lógica de predicados.

Una relación es una tabla con columnas y filas. Un SGBD sólo necesita que el usuario pueda percibir la base de datos como un conjunto de tablas. Esta percepción sólo se aplica a la estructura lógica de la base de datos, no se aplica a la estructura física de la base de datos, que se puede implementar con distintas estructuras de almacenamiento.

Un atributo es el nombre de una columna de una relación. En el modelo relacional, las relaciones se utilizan para almacenar información sobre los objetos que se representan en la base de datos. Una relación se representa gráficamente como una tabla bidimensional en la que las filas corresponden a registros individuales y las columnas corresponden a los campos o atributos de esos registros. Los atributos pueden aparecer en la relación en cualquier orden.

Por ejemplo, la información de los clientes de una empresa determinada se representa mediante la relación `CLIENTES` de la figura 2.1, que tiene columnas para los atributos `codcli` (código del cliente), `nombre` (nombre y apellidos del cliente), `dirección` (calle y número donde se ubica el cliente), `codpostal` (código postal correspondiente a la dirección del cliente) y `codpue` (código de la población del cliente). La información sobre las poblaciones se representa

mediante la relación PUEBLOS de la misma figura, que tiene columnas para los atributos codpue (código de la población), nombre (nombre de la población) y codpro (código de la provincia en que se encuentra la población).

CLIENTES

codcli	nombre	dirección	codpostal	codpue
333	Sos Carretero, Jesús	Mosen Compte, 14	12964	53596
336	Miguel Archilés, Ramón	Bernardo Mundina, 132-5	12652	07766
342	Pinel Huerta, Vicente	Francisco Sempere, 37-10	12112	07766
345	López Botella, Mauro	Avenida del Puerto, 20-1	12439	12309
348	Palau Martínez, Jorge	Raval de Sant Josep, 97-2	12401	12309
354	Murría Vinaiza, José	Ciudadela, 90-18	12990	12309
357	Huguet Peris, Juan Ángel	Calle Mestre Rodrigo, 7	12930	12309

PUEBLOS

codpue	nombre	codpro
07766	Burriana	12
12309	Castellón	12
17859	Enramona	12
46332	Soneja	12
53596	Vila-real	12

Figura 2.1: Relaciones que almacenan los datos de los clientes y sus poblaciones.

Un dominio es el conjunto de valores legales de uno o varios atributos. Los dominios constituyen una poderosa característica del modelo relacional. Cada atributo de una base de datos relacional se define sobre un dominio, pudiendo haber varios atributos definidos sobre el mismo dominio. La figura 2.2 muestra los dominios de los atributos de la relación CLIENTES.

Atributo	*Dominio*	*Descripción*	*Definición*
codcli	codli_dom	Posibles códigos de cliente.	Número hasta 5 dígitos.
nombre	nombre_dom	Nombres de personas: apellido1 apellido2, nombre.	50 caracteres.
dirección	dirección_dom	Domicilios de España: calle, número.	50 caracteres.
codpostal	codpostal_dom	Códigos postales de España.	5 caracteres.
codpue	codpue_dom	Códigos de las poblaciones de España.	5 caracteres.

Figura 2.2: Dominios de los atributos de la relación que almacena los datos de los clientes.

El concepto de dominio es importante porque permite que el usuario defina, en un lugar común, el significado y la fuente de los valores que los atributos pueden tomar. Esto hace que haya más información disponible para el sistema cuando éste va a ejecutar una operación relacional, de modo que las operaciones que son semánticamente incorrectas, se pueden evitar. Por ejemplo, no tiene sentido comparar el nombre de una calle con un número de teléfono, aunque los dos atributos sean cadenas de caracteres. Sin embargo, el importe mensual del alquiler de un inmueble no estará definido sobre el mismo dominio que

el número de meses que dura el alquiler, pero sí tiene sentido multiplicar los valores de ambos dominios para averiguar el importe total al que asciende el alquiler. Los SGBD relacionales no ofrecen un soporte completo de los dominios ya que su implementación es extremadamente compleja.

Una tupla es una fila de una relación. Los elementos de una relación son las tuplas o filas de la tabla. En la relación `CLIENTES`, cada tupla tiene cinco valores, uno para cada atributo. Las tuplas de una relación no siguen ningún orden.

El grado de una relación es el número de atributos que contiene. La relación `CLIENTES` es de grado cinco porque tiene cinco atributos. Esto quiere decir que cada fila de la tabla es una tupla con cinco valores. El grado de una relación no cambia con frecuencia.

La cardinalidad de una relación es el número de tuplas que contiene. Ya que en las relaciones se van insertando y borrando tuplas a menudo, la cardinalidad de las mismas varía constantemente.

Una base de datos relacional es un conjunto de relaciones normalizadas. Una relación está normalizada si en la intersección de cada fila con cada columna hay un solo valor.

Definiciones formales

Una *relación* R definida sobre un conjunto de *dominios* $D_1, D_2, \ldots, D_n$ consta de:

- *Cabecera:* conjunto fijo de pares *atributo:dominio*

  $$\{(A_1 : D_1), (A_2 : D_2), \ldots, (A_n : D_n)\}$$

  donde cada atributo A_j corresponde a un único dominio D_j y todos los A_j son distintos, es decir, no hay dos atributos que se llamen igual. El grado de la relación R es n.

- *Cuerpo:* conjunto variable de *tuplas*. Cada tupla es un conjunto de pares *atributo:valor*:

  $$\{(A_1 : v_{i1}), (A_2 : v_{i2}), \ldots, (A_n : v_{in})\}$$

  con $i = 1, 2, \ldots, m$, donde m es la cardinalidad de la relación R. En cada par $(A_j : v_{ij})$ se tiene que $v_{ij} \in D_j$.

A continuación se muestra la cabecera de la relación `CLIENTES` de la figura 2.1, y una de sus tuplas.

```
{ (codcli:codcli_dom), (nombre:nombre_dom),
  (dirección:dirección_dom), (codpostal:codpostal_dom),
  (codpue:codpue_dom) }
```

```
{ (codcli:333), (nombre:Sos Carretero, Jesús),
  (dirección:Mosen Compte, 14), (codpostal:12964),
  (codpue:53596) }
```

Este conjunto de pares no está ordenado, por lo que la tupla anterior y la siguiente son la misma:

```
{ (nombre:Sos Carretero, Jesús), (codpostal:12964),
  (codcli:333), (dirección:Mosen Compte, 14),
  (codpue:53596) }
```

Las relaciones se suelen representar gráficamente mediante tablas. Los nombres de las columnas corresponden a los nombres de los atributos, y las filas son cada una de las tuplas de la relación. Los valores que aparecen en cada una de las columnas pertenecen al conjunto de valores del dominio sobre el que está definido el atributo correspondiente.

2.2.2. Propiedades de las relaciones

Las relaciones tienen las siguientes características:

- Cada relación tiene un nombre, y éste es distinto del nombre de todas las demás.
- Los dominios sobre los que se definen los atributos son escalares, por lo que los valores de los atributos son atómicos. De este modo, en cada tupla, cada atributo toma un solo valor. Se dice que las relaciones están *normalizadas*.
- No hay dos atributos que se llamen igual.
- El orden de los atributos no importa: los atributos no están ordenados.
- Cada tupla es distinta de las demás: no hay tuplas duplicadas.
- El orden de las tuplas no importa: las tuplas no están ordenadas.

2.2.3. Tipos de relaciones

En un SGBD relacional hay dos tipos de relaciones:

- *Relaciones base.* Son relaciones reales que tienen nombre, y forman parte directa de la base de datos almacenada. Se dice que las relaciones base son relaciones autónomas.
- *Vistas.* También denominadas relaciones virtuales, son relaciones con nombre y derivadas (no autónomas). Que son derivadas significa que se obtienen a partir de otras relaciones; se representan mediante su definición en términos de esas otras relaciones. Las vistas no poseen datos almacenados propios, los datos que contienen corresponden a datos almacenados en relaciones base.

2.2.4. Claves

Ya que en una relación no hay tuplas repetidas, éstas se pueden distinguir unas de otras, es decir, se pueden identificar de modo único. La forma de identificarlas es mediante los valores de sus atributos. Se denomina *superclave* a un atributo o conjunto de atributos que identifican de modo único las tuplas de una relación. Se denomina *clave candidata* a una superclave en la que ninguno de sus subconjuntos es una superclave de la relación. El atributo o conjunto de atributos K de la relación R es una clave candidata para R si, y sólo si, satisface las siguientes propiedades:

- *Unicidad:* nunca hay dos tuplas en la relación R con el mismo valor de K.
- *Irreducibilidad (minimalidad):* ningún subconjunto de K tiene la propiedad de unicidad, es decir, no se pueden eliminar componentes de K sin destruir la unicidad.

Cuando una clave candidata está formada por más de un atributo, se dice que es una *clave compuesta*. Una relación puede tener varias claves candidatas. Por ejemplo, en la relación `PUEBLOS` de la figura 2.1, el atributo `nombre` no es una clave candidata ya que hay pueblos en España con el mismo nombre que se encuentran en distintas provincias. Sin embargo, se ha asignado un código único a cada población, por lo que el atributo `codpue` sí es una clave candidata de la relación `PUEBLOS`. También es una clave candidata de esta relación la pareja formada por los atributos `nombre` y `codpro`, ya que no hay dos poblaciones en la misma provincia que tengan el mismo nombre.

Para identificar las claves candidatas de una relación no hay que fijarse en un estado u ocurrencia de la base de datos. El hecho de que en un momento dado no haya duplicados para un atributo o conjunto de atributos, no garantiza que los duplicados no sean posibles. Sin embargo, la presencia de duplicados

en un estado de la base de datos sí es útil para demostrar que cierta combinación de atributos no es una clave candidata. El único modo de identificar las claves candidatas es conociendo el significado real de los atributos, ya que esto permite saber si es posible que aparezcan duplicados. Sólo usando esta información semántica se puede saber con certeza si un conjunto de atributos forman una clave candidata. Por ejemplo, viendo la ocurrencia anterior de la relación `CLIENTES` se podría pensar que el atributo `nombre` es una clave candidata. Pero ya que este atributo es el nombre de un cliente y es posible que haya dos clientes con el mismo nombre, el atributo no es una clave candidata.

Se denomina *clave primaria* de una relación a aquella clave candidata que se escoge para identificar sus tuplas de modo único. Ya que una relación no tiene tuplas duplicadas, siempre hay una clave candidata y, por lo tanto, la relación siempre tiene clave primaria. En el peor caso, la clave primaria estará formada por todos los atributos de la relación, pero normalmente habrá un pequeño subconjunto de los atributos que haga esta función.

Las claves candidatas que no son escogidas como clave primaria son denominadas *claves alternativas*. Por ejemplo, la clave primaria de la relación `PUEBLOS` es el atributo `codpue`, siendo la pareja formada por `nombre` y `codpro` una clave alternativa. En la relación `CLIENTES` sólo hay una clave candidata que es el atributo `codcli`, por lo que esta clave candidata es la clave primaria.

Una *clave ajena* es un atributo o un conjunto de atributos de una relación cuyos valores coinciden con los valores de la clave primaria de alguna otra relación (puede ser la misma). Las claves ajenas representan *relaciones entre datos*. Por ejemplo, el atributo `codpue` de `CLIENTES` relaciona a cada cliente con su población. Este atributo en `CLIENTES` es una clave ajena cuyos valores hacen referencia al atributo `codpue` de `PUEBLOS` (su clave primaria). Se dice que un valor de clave ajena representa una *referencia* a la tupla que contiene el mismo valor en su clave primaria (*tupla referenciada*).

Si nos fijamos en los datos de la figura 2.1, para conocer el nombre de la población del cliente con `codcli` = `333`, debemos seguir la clave ajena `codpue` que aparece en la tupla de dicho cliente y que tiene el valor `53596`. Seguir la referencia que implica la clave ajena conlleva visitar la relación `PUEBLOS` y localizar la fila que tiene el valor `53596` en su clave primaria. Nótese que, en este ejemplo, la clave ajena tiene el mismo nombre que la clave primaria a la que hace referencia. Esto no es un requisito, las claves ajenas no precisan tener el mismo nombre que la clave primaria a la que referencian; sin embargo, si se utilizan los mismos nombres (o nombres compuestos derivados de los mismos) es más fácil reconocer las claves ajenas.

Al hablar de claves primarias y de claves ajenas es importante darse cuenta de que los valores de una clave primaria no se pueden repetir, mientras que no sucede lo mismo con las claves ajenas que le hacen referencia. Así, en las tablas de la figura 2.1 no es posible encontrar dos tuplas con el mismo valor en `PUEBLOS.codpue` (cada población debe aparecer en la relación una sola vez), pero sí es posible encontrar varias tuplas con el mismo valor en

CLIENTES.codpue, ya que es posible que haya varios clientes que se ubiquen en la misma población.

2.3. Esquema de una base de datos relacional

Una base de datos relacional es un conjunto de relaciones. Para representar el esquema de una base de datos relacional se debe dar el nombre de sus relaciones, los atributos de éstas, los dominios sobre los que se definen estos atributos, las claves primarias y las claves ajenas.

El esquema de la base de datos de la empresa con la que trabajaremos en este libro es el siguiente:

```
   CLIENTES(codcli, nombre, dirección, codpostal, codpue)
 VENDEDORES(codven, nombre, dirección, codpostal, codpue, codjefe)
    PUEBLOS(codpue, nombre, codpro)
 PROVINCIAS(codpro, nombre)
  ARTÍCULOS(codart, descrip, precio, stock, stock_min, dto)
   FACTURAS(codfac, fecha, codcli, codven, iva, dto)
 LÍNEAS_FAC(codfac, línea, cant, codart, precio, dto)
```

En el esquema anterior, los nombres de las relaciones aparecen seguidos de los nombres de los atributos encerrados entre paréntesis. Las claves primarias son los atributos subrayados. Las claves ajenas se representan mediante los siguientes *diagramas referenciales:*

CLIENTES	$\xrightarrow{\texttt{codpue}}$	PUEBLOS	:	Población del cliente.
VENDEDORES	$\xrightarrow{\texttt{codpue}}$	PUEBLOS	:	Población del vendedor.
VENDEDORES	$\xrightarrow{\texttt{codjefe}}$	VENDEDORES	:	Jefe del vendedor.
PUEBLOS	$\xrightarrow{\texttt{codpro}}$	PROVINCIAS	:	Provincia en la que se encuentra la población.
FACTURAS	$\xrightarrow{\texttt{codcli}}$	CLIENTES	:	Cliente al que pertenece la factura.
FACTURAS	$\xrightarrow{\texttt{codven}}$	VENDEDORES	:	Vendedor que ha realizado la venta.
LÍNEAS_FAC	$\xrightarrow{\texttt{codfac}}$	FACTURAS	:	Factura en la que se encuentra la línea.
LÍNEAS_FAC	$\xrightarrow{\texttt{codart}}$	ARTÍCULOS	:	Artículo que se compra en la línea de factura.

La tabla PROVINCIAS almacena información sobre las provincias de España. De cada provincia se almacena su nombre (nombre) y un código que la identifica (codpro). La tabla PUEBLOS contiene los nombres (nombre) de los pueblos de España. Cada pueblo se identifica por un código que es único (codpue) y tiene una referencia a la provincia a la que pertenece (codpro). La tabla CLIENTES contiene los datos de los clientes: código que identifica a

cada uno (`codcli`), nombre y apellidos (`nombre`), calle y número (`dirección`), código postal (`codpostal`) y una referencia a su población (`codpue`). La tabla `VENDEDORES` contiene los datos de los vendedores de la empresa: código que identifica a cada uno (`codven`), nombre y apellidos (`nombre`), calle y número (`dirección`), código postal (`codpostal`), una referencia a su población (`codpue`) y una referencia al vendedor del que depende (`codjefe`), si es el caso. En la tabla `ARTÍCULOS` se tiene el código que identifica a cada artículo (`codart`), su descripción (`descrip`), el precio de venta actual (`precio`), el número de unidades del artículo que hay en el almacén (`stock`), la cantidad mínima que se desea mantener almacenada (`stock_min`) y, si el artículo está en oferta, el descuento (`dto`) que se debe aplicar cuando se venda. La tabla `FACTURAS` contiene las cabeceras de las facturas correspondientes a las compras realizadas por los clientes. Cada factura tiene un código único (`codfac`), la fecha en que se ha realizado (`fecha`), así como el IVA (`iva`) y el descuento que se le ha aplicado (`dto`). Cada factura hace referencia al cliente al que pertenece (`codcli`) y al vendedor que la ha realizado (`codven`). Las líneas de cada factura se encuentran en la tabla `LÍNEAS_FAC`, identificándose cada una por el número de línea que ocupa dentro de la factura (`codfac, línea`). En cada una de ellas se especifica la cantidad de unidades (`cant`) del artículo que se compra (`codart`), el precio de venta por unidad (`precio`) y el descuento que se aplica sobre dicho precio (`dto`), si es que el artículo estaba en oferta cuando se vendió.

A continuación se muestra un estado de la base de datos cuyo esquema se acaba de definir.

CLIENTES

codcli	nombre	dirección	codpostal	codpue
333	Sos Carretero, Jesús	Mosen Compte, 14	12964	53596
336	Miguel Archilés, Ramón	Bernardo Mundina, 132-5	12652	07766
342	Pinel Huerta, Vicente	Francisco Sempere, 37-10	12112	07766
345	López Botella, Mauro	Avenida del Puerto, 20-1	12010	12309
348	Palau Martínez, Jorge	Raval de Sant Josep, 97-2	12003	12309
354	Murría Vinaiza, José	Ciudadela, 90-18	12003	12309
357	Huguet Peris, Juan Ángel	Calle Mestre Rodrigo, 7	12100	12309

VENDEDORES

codven	nombre	dirección	codpostal	codpue	codjefe
5	Guillén Vilar, Natalia	Sant Josep, 110	12597	53596	105
105	Poy Omella, Paloma	Sanchis Tarazona, 103-1	12257	46332	
155	Rubert Cano, Diego	Benicarló Residencial, 154	12425	17859	5
455	Agost Tirado, Jorge	Pasaje Peñagolosa, 21-19	12914	53596	5

PUEBLOS

codpue	nombre	codpro
07766	Burriana	12
12309	Castellón	12
17859	Enramona	12
46332	Soneja	12
53596	Vila-real	12

PROVINCIAS

codpro	nombre
03	Alicante
12	Castellón
46	Valencia

ARTÍCULOS

codart	descrip	precio	stock	stock_min	dto
IM3P32V	Interruptor magnetotérmico 4p, 2	27.01	1	1	
im4P10L	Interruptor magnetotérmico 4p, 4	32.60	1	1	15
L14340	Bases de fusibles cuchillas T0	0.51	3	3	
L17055	Bases de fusible cuchillas T3	7.99	3	3	
L76424	Placa 2 E. legrand serie mosaic	2.90	5	2	
L85459	Tecla legrand marfil	2.80	0	4	
L85546	Tecla difusores legrand bronce	1.05	13	5	5
L92119	Portalámparas 14 curvo	5.98	2	1	
ME200	Marco Bjc Ibiza 2 elementos	13.52	1	1	
N5072	Pulsador luz piloto Niessen trazo	1.33	11	2	
N8017BA	Reloj Orbis con reserva de cuerda	3.40	7	4	
P605	Caja 1 elem. plastimetal	1.65	16	9	
P695	Interruptor rotura brusca 100 A M	13.22	1	1	
P924	Interruptor marrón dec. con visor	2.39	8	3	
REF1X20	Regleta fluorescente 1x36 bajo F	8.71	1	1	
S3165136	Bloque emergencia Satf 150 L	4.81	6	3	
T4501	Tubo empotrar 100	2.98	0	5	
TE7200	Doble conmutador Bjc Ibiza blanco	13.22	1	1	
TFM16	Curva tubo hierro 11	0.33	23	13	
TH11	Curva tubo hierro 29	1.42	20	3	
THC21	Placa mural Felmax	1.56	1	1	
ZNCL	Base T,t lateral Ticino S, Tekne	41.71	1	1	10

FACTURAS

codfac	fecha	codcli	codven	iva	dto
6643	16/07/2010	333	105	18	10
6645	16/07/2010	336	105	0	20
6654	31/07/2010	357	155	8	0
6659	08/08/2010	342	5	0	0
6680	10/09/2010	348	455	8	0
6723	06/11/2010	342	5	18	0
6742	17/12/2010	333	105	8	20

LÍNEAS_FAC

codfac	linea	cant	codart	precio	dto
6643	1	6	L14340	0.51	20
6643	2	1	N5072	1.33	0
6643	3	2	P695	13.22	0
6645	1	10	ZNCL	41.71	0
6645	2	6	N8017BA	3.40	0
6645	3	3	TE7200	13.22	0
6645	4	4	L92119	5.98	0
6654	1	6	REF1X20	8.71	50
6659	1	8	THC21	1.56	0
6659	2	12	L17055	7.99	25
6659	3	9	L76424	2.90	0
6680	1	12	T4501	2.98	0
6680	2	11	im4P10L	32.60	0
6723	1	5	L85459	2.80	5
6742	1	9	ME200	13.52	0
6742	2	8	S3165136	4.81	5

2.4. Reglas de integridad

Una vez definida la estructura de datos del modelo relacional, pasamos a estudiar las reglas de integridad que los datos almacenados en dicha estructura deben cumplir para garantizar que son correctos.

Al definir cada atributo sobre un dominio se impone una restricción sobre el conjunto de valores permitidos para cada atributo. A este tipo de restricciones se les denomina *restricciones de dominios*. Hay además dos reglas de integridad muy importantes que son restricciones que se deben cumplir en todas las bases de datos relacionales y en todos sus estados (las reglas se deben cumplir todo el tiempo). Estas reglas son la *regla de integridad de entidades* y la *regla de integridad referencial*. Antes de definirlas, es preciso conocer el concepto de *nulo*.

2.4.1. Nulos

Cuando en una tupla un atributo es desconocido, se dice que es *nulo*. Un nulo no representa el valor cero ni la cadena vacía ya que éstos son valores que tienen significado. El nulo implica ausencia de información, bien porque al insertar la tupla se desconocía el valor del atributo, o bien porque para dicha tupla el atributo no tiene sentido.

Ya que los nulos no son valores, deben tratarse de modo diferente, lo que causa problemas de implementación. De hecho, no todos los SGBD relacionales soportan los nulos.

2.4.2. Regla de integridad de entidades

La primera regla de integridad se aplica a las claves primarias de las relaciones base: *ninguno de los atributos que componen la clave primaria puede ser nulo.*

Por definición, una clave primaria es una clave irreducible que se utiliza para identificar de modo único las tuplas. Que es irreducible significa que ningún subconjunto de la clave primaria sirve para identificar las tuplas de modo único. Si se permitiera que parte de la clave primaria fuera nula, se estaría diciendo que no todos sus atributos son necesarios para distinguir las tuplas, con lo que se estaría contradiciendo la irreducibilidad.

Nótese que esta regla sólo se aplica a las relaciones base y a las claves primarias, no a las claves alternativas.

2.4.3. Regla de integridad referencial

La segunda regla de integridad se aplica a las claves ajenas: *si en una relación hay alguna clave ajena, sus valores deben coincidir con valores de la clave primaria a la que hace referencia, o bien, deben ser completamente nulos.*

En la base de datos presentada en el apartado anterior hay ocho claves ajenas que mantienen relacionada la información almacenada en las tablas. Así, a través de la clave ajena `FACTURAS.codcli` se puede conocer los datos personales del cliente al que pertenece una determinada factura buscando en la relación `CLIENTES` la tupla en cuya clave primaria aparece el valor de `codcli` al que se hace referencia en la factura. El nombre de la población del cliente se podrá conocer siguiendo la clave ajena `CLIENTES.codpue` y, una vez localizada la población con dicho `codpue` en `PUEBLOS`, se podrá acceder al nombre de su provincia siguiendo la clave ajena `PUEBLOS.codpro`.

Pues bien, la regla de integridad referencial exige que los valores que aparecen en la clave ajena `FACTURAS.codcli` aparezcan como clave primaria en `CLIENTES`. De ese modo, todas las facturas corresponderán a clientes cuyos datos se encuentran en la base de datos. Del mismo modo, la regla exige que los valores de `CLIENTES.codpue` aparezcan en la clave primaria de `PUEBLOS` y que los valores de `PUEBLOS.codpro` aparezcan en la clave primaria de `PROVINCIAS`.

La regla de integridad referencial se enmarca en términos de estados de la base de datos: indica lo que es un estado ilegal, pero no dice cómo puede evitarse. Por lo tanto, una vez establecida la regla, hay que plantearse qué hacer si estando en un estado legal, llega una petición para realizar una operación que conduce a un estado ilegal. Existen dos opciones: *rechazar* o *aceptar* la operación y realizar operaciones adicionales compensatorias que conduzcan a un estado legal.

Para hacer respetar la integridad referencial se debe contestar, para cada clave ajena, a las tres preguntas que se plantean a continuación y que determinarán su comportamiento:

- *Regla de los nulos:* «¿Tiene sentido que la clave ajena acepte nulos?»
- *Regla de borrado:* «¿Qué ocurre si se intenta borrar la tupla referenciada por la clave ajena?»
  - *Restringir:* no se permite borrar la tupla referenciada.
  - *Propagar:* se borra la tupla referenciada y se propaga el borrado a las tuplas que la referencian mediante la clave ajena.
  - *Anular:* se borra la tupla referenciada y las tuplas que la referenciaban ponen a nulo la clave ajena (sólo si acepta nulos).
  - *Valor por defecto:* se borra la tupla referenciada y las tuplas que la referenciaban ponen en la clave ajena el valor por defecto establecido para la misma.

- *Regla de modificación:* «¿Qué ocurre si se intenta modificar el valor de la clave primaria de la tupla referenciada por la clave ajena?»

  - *Restringir:* no se permite modificar el valor de la clave primaria de la tupla referenciada.
  - *Propagar:* se modifica el valor de la clave primaria de la tupla referenciada y se propaga la modificación a las tuplas que la referencian, mediante la clave ajena.
  - *Anular:* se modifica la tupla referenciada y las tuplas que la referenciaban ponen a nulo la clave ajena (sólo si acepta nulos).
  - *Valor por defecto:* se modifica la tupla referenciada y las tuplas que la referenciaban ponen en la clave ajena el valor por defecto establecido para la misma.

Así, en el caso del esquema de la base de datos presentada en el apartado anterior, deberemos determinar las reglas de comportamiento para cada clave ajena. Por ejemplo, para la clave ajena `FACTURAS.codcli` se ha escogido el siguiente comportamiento:

- *Regla de los nulos:* la clave ajena acepta nulos, por lo que es posible encontrar facturas cuyo cliente se ignore (esto se ha decidido así porque lo impone un requisito del usuario).

- *Regla de borrado:* anular. Cuando se elimine un cliente de la base de datos y se proceda a borrarlo de la relación `CLIENTES`, se deberán anular todas las referencias que hubiera desde `FACTURAS.codcli`. De este modo, todas las facturas que tenía ese cliente pasarán a tener un nulo en el código del cliente.

- *Regla de modificación:* propagar. En caso de que se modifique el código a un cliente (quizá porque el sistema de codificación se cambie por parte de la empresa), todas las facturas de dicho cliente actualizarán el valor de `FACTURAS.codcli` para continuar haciendo referencia a la misma tupla.

Del mismo modo, se deberá escoger reglas para el resto de las claves ajenas de la base de datos. Una vez establecidas todas las reglas, el sistema se comportará de manera coherente obedeciendo a todas las reglas impuestas. Por ejemplo, si la regla de borrado para `LÍNEAS_FAC.codfac` es propagar y la regla de borrado para `FACTURAS.codven` es restringir, cuando se borre una tupla en `FACTURAS` se propagará el borrado a `LÍNEAS_FAC`, y se borrarán todas las líneas de la factura referenciada. Sin embargo, cuando se intente borrar la tupla de un vendedor que aparezca en alguna factura, la regla impuesta sobre `FACTURAS.codven` rechazará el borrado del vendedor y no se procederá al borrado ni de sus facturas, ni de las líneas de factura de éstas. Lo que sí será posible es el borrado de un vendedor cuyo código no aparezca en ninguna factura.

2.4.4. Reglas de negocio

Además de las dos reglas de integridad anteriores, es posible que sea necesario imponer ciertas restricciones específicas sobre los datos que forman parte de la estrategia de funcionamiento de la empresa. A estas reglas se las denomina *reglas de negocio*.

Por ejemplo, si en cada oficina de una determinada empresa sólo puede haber hasta veinte empleados, el SGBD debe dar la posibilidad al usuario de definir una regla al respecto y debe hacerla respetar. En este caso, no debería permitir dar de alta a un empleado en una oficina que ya tiene los veinte permitidos. No todos los SGBD relacionales permiten definir este tipo de restricciones y hacerlas respetar.

Capítulo 3

Lenguajes relacionales

Introducción y objetivos

La tercera parte de un modelo de datos es la de la manipulación de los datos. En este capítulo se presentan el álgebra relacional y el cálculo relacional, definidos por E. F. Codd como la base de los lenguajes relacionales.

Al finalizar este capítulo, el estudiante debe ser capaz de:

- Emplear los operadores del álgebra relacional para responder a cualquier consulta de datos.
- Emplear los operadores del cálculo relacional orientado a tuplas para responder a consultas de datos que no requieran operaciones de resumen.
- Describir la diferencia entre el cálculo relacional orientado a tuplas y el cálculo relacional orientado a dominios.
- Enumerar otros lenguajes relacionales distintos al álgebra y el cálculo relacional.

3.1. Manejo de datos

Son varios los lenguajes utilizados por los SGBD relacionales para manejar las relaciones. Algunos de ellos son *procedurales*, lo que quiere decir que el usuario indica al sistema exactamente cómo debe manipular los datos. Otros son *no procedurales*, que significa que el usuario indica qué datos necesita, en lugar de establecer cómo deben obtenerse. Se puede decir que el álgebra relacional es un lenguaje procedural de alto nivel, mientras que el cálculo relacional es un lenguaje no procedural. Sin embargo, ambos lenguajes son equivalentes: para cada expresión del álgebra, se puede encontrar una expresión equivalente en el cálculo, y viceversa.

El álgebra relacional (o el cálculo relacional) se utiliza para medir la potencia de los lenguajes relacionales. Si un lenguaje permite obtener cualquier

relación que se pueda derivar mediante el álgebra relacional, se dice que es *relacionalmente completo*. La mayoría de los lenguajes relacionales son relacionalmente completos, pero tienen más potencia que el álgebra o el cálculo porque se les han añadido operadores especiales.

Tanto el álgebra como el cálculo son lenguajes formales no muy amigables; sin embargo, es conveniente estudiarlos porque sirven para ilustrar las operaciones básicas que todo lenguaje de manejo de datos debe ofrecer. Además, han sido la base para otros lenguajes relacionales de manejo de datos de más alto nivel.

3.2. Álgebra relacional

El álgebra relacional es un lenguaje formal con una serie de operadores que trabajan sobre una o varias relaciones para obtener otra relación resultado, sin que cambien las relaciones originales. Tanto los operandos como los resultados son relaciones, por lo que la salida de una operación puede ser la entrada de otra operación. Esto permite anidar expresiones del álgebra, del mismo modo que se pueden anidar las expresiones aritméticas. A esta propiedad se le denomina *clausura*: las relaciones son cerradas bajo el álgebra, del mismo modo que los números son cerrados bajo las operaciones aritméticas.

En este apartado se describen, en primer lugar, los ocho operadores originalmente propuestos por Codd, y después se estudian algunos operadores adicionales que añaden potencia al lenguaje.

De los ocho operadores, sólo hay cinco que son fundamentales: *restricción*, *proyección*, *producto cartesiano*, *unión* y *diferencia*. Los operadores fundamentales permiten realizar la mayoría de las operaciones de obtención de datos. Los operadores no fundamentales son la *concatenación (join)*, la *intersección* y la *división*, que se pueden expresar a partir de los cinco operadores fundamentales.

La restricción y la proyección son operaciones *unarias* porque operan sobre una sola relación. El resto de las operaciones son *binarias* porque trabajan sobre pares de relaciones. En las definiciones que se presentan a continuación, se supone que R y S son dos relaciones cuyos atributos son $A = (a_1, a_2, \ldots, a_N)$ y $B = (b_1, b_2, \ldots, b_M)$ respectivamente.

A continuación, se presentan los operadores del álgebra relacional, mostrando su uso mediante breves ejemplos. Todos estos ejemplos están basados en el esquema de la base de datos relacional presentada en el capítulo anterior (apartado 2.3).

Restricción: `R WHERE condición`

La restricción, también denominada *selección*, opera sobre una sola relación `R` y da como resultado otra relación cuyas tuplas son las tuplas de `R` que satisfacen la condición especificada. Esta condición es una comparación en la que aparece al menos un atributo de `R`, o una combinación booleana de varias de estas comparaciones.

Ejemplo 3.1 *Obtener todos los artículos que tienen un precio superior a 10 €.*

Expresión del álgebra relacional que obtiene los datos especificados:

```
ARTICULOS WHERE precio>10
```

Resultado:

codart	descrip	precio	stock	stock_min	dto
IM3P32V	Interruptor magnetotérmico 4p, 2	27.01	1	1	
im4P10L	Interruptor magnetotérmico 4p, 4	32.60	1	1	15
ME200	Marco Bjc Ibiza 2 elementos	13.52	1	1	
P695	Interruptor rotura brusca 100 A M	13.22	1	1	
TE7200	Doble conmutador Bjc Ibiza blanco	13.22	1	1	
ZNCL	Base T,t lateral Ticino S, Tekne	41.71	1	1	10

Ejemplo 3.2 *Obtener los artículos cuyo stock es de menos de 5 unidades y además se ha quedado al mínimo o por debajo.*

Expresión del álgebra relacional que obtiene los datos especificados:

```
ARTÍCULOS WHERE stock<5 AND stock<stock_min
```

Resultado:

codart	descrip	precio	stock	stock_min	dto
IM3P32V	Interruptor magnetotérmico 4p, 2	27.01	1	1	
im4P10L	Interruptor magnetotérmico 4p, 4	32.60	1	1	15
L14340	Bases de fusibles cuchillas T0	0.51	3	3	
L17055	Bases de fusible cuchillas T3	7.99	3	3	
L85459	Tecla Legrand marfil	2.80	0	4	
...	...	...	...	...	

Proyección: `R[`a_i`, ..., `a_k`]`

La proyección opera sobre una sola relación `R` y da como resultado otra relación que contiene un subconjunto vertical de `R`, extrayendo los valores de los atributos especificados y eliminando duplicados.

Ejemplo 3.3 *Obtener un listado de vendedores mostrando su código, su nombre y su código postal.*

Expresión del álgebra relacional que obtiene los datos especificados:

```
VENDEDORES[codven,nombre,codpostal]
```

Resultado:

codven	nombre	codpostal
5	Guillén Vilar, Natalia	12597
105	Poy Omella, Paloma	12257
155	Rubert Cano, Diego	12425
455	Agost Tirado, Jorge	12914

Ejemplo 3.4 *Obtener los códigos de las poblaciones donde hay clientes.*

Expresión del álgebra relacional que obtiene los datos especificados:

```
CLIENTES[codpue]
```

Resultado:

codpue
53596
07766
12309

Producto cartesiano: `R TIMES S`
El producto cartesiano obtiene una relación cuyas tuplas están formadas por la concatenación de todas las tuplas de `R` con todas las tuplas de `S`.

La restricción y la proyección son operaciones que permiten extraer información de una sola relación. Habrá casos en que sea necesario combinar la información de varias relaciones. El producto cartesiano multiplica dos relaciones, definiendo una nueva relación que tiene todos los pares posibles de tuplas de las dos relaciones. Si la relación `R` tiene p tuplas y n atributos y la relación `S` tiene q tuplas y m atributos, la relación resultado tendrá $p * q$ tuplas y $n + m$ atributos. Ya que es posible que haya atributos con el mismo nombre en las dos relaciones, el nombre de la relación se antepondrá al del atributo en este caso para que los nombres de los atributos sigan siendo únicos en la relación resultado.

Una vez realizado el producto cartesiano de dos relaciones, se puede realizar una restricción que elimine aquellas tuplas cuya información no esté relacionada, como se muestra en el siguiente ejemplo.

Ejemplo 3.5 *Obtener los nombres de las poblaciones en las que hay clientes.*

Expresión del álgebra relacional que obtiene los datos especificados:

```
(CLIENTES[codpue] TIMES PUEBLOS)
     WHERE CLIENTES.codpue = PUEBLOS.codpue
```

Resultado:

CLIENTES.codpue	PUEBLOS.codpue	nombre	codpro
53596	53596	Vila-real	12
07766	07766	Burriana	12
12309	12309	Castellón	12

La combinación del producto cartesiano y la restricción del modo en que se acaba de realizar, se puede reducir a la operación de *concatenación* (`JOIN`) que se presenta más adelante.

> **Unión:** `R UNION S`
> La unión de dos relaciones `R` y `S`, con P y Q tuplas respectivamente, es otra relación que tiene como mucho $P + Q$ tuplas siendo éstas las tuplas que se encuentran en `R` o en `S` o en ambas relaciones a la vez. Para poder realizar esta operación, `R` y `S` deben ser compatibles para la unión.

Se dice que dos relaciones son *compatibles para la unión* si ambas tienen la misma cabecera, es decir, si tienen el mismo número de atributos y éstos se encuentran definidos sobre los mismos dominios en ambas tablas respectivamente. En muchas ocasiones será necesario realizar proyecciones para hacer que dos relaciones sean compatibles para la unión.

Ejemplo 3.6 *Obtener un listado de los códigos de las poblaciones donde hay clientes o vendedores.*

Expresión del álgebra relacional que obtiene los datos especificados:

```
CLIENTES[codpue] UNION VENDEDORES[codpue]
```

Resultado:

codpue
53596
07766
12309
46332
17859

> **Diferencia:** `R EXCEPT S`
> La diferencia obtiene una relación que tiene las tuplas que se encuentran en `R` y no se encuentran en `S`. Para realizar esta operación, `R` y `S` deben ser compatibles para la unión.

Ejemplo 3.7 *Obtener un listado de las poblaciones en donde hay clientes y no hay vendedores.*

Expresión del álgebra relacional que obtiene los datos especificados:

```
CLIENTES[codpue] EXCEPT VENDEDORES[codpue]
```

Resultado:

codpue
07766
12309

Concatenación (Join): `R JOIN S`
La concatenación de dos relaciones `R` y `S` obtiene como resultado una relación cuyas tuplas son todas las tuplas de `R` concatenadas con todas las tuplas de `S` que en los atributos comunes (aquellos que se llaman igual) tienen los mismos valores. Estos atributos comunes aparecen una sola vez en el resultado.

Ejemplo 3.8 *Obtener los datos de las poblaciones en las que hay clientes.*

Expresión del álgebra relacional que obtiene los datos especificados:

```
CLIENTES[codpue] JOIN PUEBLOS
```

Esta expresión obtiene el mismo resultado que la expresión final del ejemplo 3.5, ya que la operación de concatenación es, en realidad, un producto cartesiano y una restricción de igualdad sobre los atributos comunes.

Concatenación externa (Outer-join): `R LEFT OUTER JOIN S`
La concatenación externa por la izquierda es una concatenación en la que las tuplas de `R` (que se encuentra a la izquierda en la expresión) que no tienen valores en común con ninguna tupla de `S`, también aparecen en el resultado.

Ejemplo 3.9 *Obtener un listado de todos los clientes (código y nombre) y las facturas que se les han realizado. Si no tienen facturas también deben aparecer en el resultado.*

Expresión del álgebra relacional que obtiene los datos especificados:

```
CLIENTES[codcli,nombre] LEFT OUTER JOIN FACTURAS
```

Resultado:

codfac	fecha	codcli	nombre	codven	iva	dto
6643	16/07/2010	333	Sos Carretero, Jesús	105	18	10
6645	16/07/2010	336	Miguel Archilés, Ramón	105	0	20
6654	31/07/2010	357	Huguet Peris, Juan Ángel	155	8	0
6659	08/08/2010	342	Pinel Huerta, Vicente	5	0	0
6680	10/09/2010	348	Palau Martínez, Jorge	455	8	0
6723	06/11/2010	342	Pinel Huerta, Vicente	5	18	0
6742	17/12/2010	333	Sos Carretero, Jesús	105	8	20
		345	López Botella, Mauro			
		354	Murría Vinaiza, José			

La expresión S RIGHT OUTER JOIN R es equivalente a R LEFT OUTER JOIN S. Cuando en ambas relaciones hay tuplas que no se pueden concatenar y se desea que en el resultado aparezcan también todas estas tuplas (tanto las de una relación como las de la otra), se puede utilizar la *concatenación externa completa*: R FULL OUTER JOIN S.

Intersección: R INTERSECT S
La intersección obtiene como resultado una relación que contiene las tuplas de R que también se encuentran en S. Para realizar esta operación, R y S deben ser compatibles para la unión.

La intersección se puede expresar en términos de diferencias:

```
R INTERSECT S = R EXCEPT (R EXCEPT S)
```

División: R DIVIDE BY S
Suponiendo que la cabecera de R es el conjunto de atributos A y que la cabecera de S es el conjunto de atributos B, tales que B es un subconjunto de A, y si C = A - B (los atributos de R que no están en S), la división obtiene una relación cuya cabecera es el conjunto de atributos C y que contiene las tuplas de R que están acompañadas de todas las tuplas de S.

Ejemplo 3.10 *Obtener clientes que han realizado compras a todos los vendedores.*

Expresión del álgebra relacional que obtiene los datos especificados:

```
FACTURAS[codcli,codven] DIVIDE BY VENDEDORES[codven]
```

Además de las operaciones que Codd incluyó en el álgebra relacional, otros autores han aportado otras operaciones para dar más potencia al lenguaje. Es de especial interés la *agrupación* (también denominada *resumen*) que añade capacidad computacional al álgebra.

Agrupación: SUMMARIZE R GROUP BY(a$_i$,...,a$_k$) ADD cálculo AS atributo
Esta operación agrupa las tuplas de R que tienen los mismos valores en los atributos especificados y realiza un cálculo sobre los grupos obtenidos. La relación resultado tiene como cabecera los atributos por los que se ha agrupado y el cálculo realizado, al que se da el nombre especificado en atributo.

Los cálculos que se pueden realizar sobre los grupos de filas son: suma de los valores de un atributo (SUM(a$_p$)), media de los valores de un atributo (AVG(a$_p$)), máximo y mínimo de los valores de un atributo (MAX(a$_p$), MIN(a$_p$)) y número de tuplas en el grupo (COUNT(*)). La relación resultado tendrá tantas filas como grupos se hayan obtenido.

Ejemplo 3.11 *Obtener el número de artículos (unidades en total) de cada factura.*

Expresión del álgebra relacional que obtiene los datos especificados:

```
SUMMARIZE LÍNEAS_FAC GROUP BY(codfac)
                     ADD SUM(cant) AS cant_total
```

Resultado:

codfac	cant_total
6643	9
6645	23
6654	6
6659	29
6680	23
6723	5
6742	17

3.3. Cálculo relacional

El álgebra relacional y el cálculo relacional son formalismos diferentes que representan distintos estilos de expresión del manejo de datos en el ámbito del modelo relacional. El álgebra relacional proporciona una serie de operaciones que se pueden usar para indicar al sistema cómo *construir* la relación deseada a partir de las relaciones de la base de datos. El cálculo relacional proporciona una notación para formular la *definición* de la relación deseada en términos de las relaciones de la base de datos.

El cálculo relacional toma su nombre del *cálculo de predicados*, que es una rama de la lógica. Hay dos tipos de cálculo relacional, el *orientado a tuplas*, propuesto por Codd, y el *orientado a dominios*, propuesto por otros autores. El estudio del cálculo relacional se hará aquí mediante definiciones informales. Las definiciones formales se pueden encontrar en la bibliografía.

En el cálculo de predicados (lógica de primer orden), un *predicado* es una función con argumentos que se puede evaluar a verdadero o falso. Cuando los argumentos se sustituyen por valores, la función lleva a una expresión denominada *proposición*, que puede ser verdadera o falsa. Por ejemplo, las frases «Paloma Poy es una vendedora de la empresa» y «Paloma Poy es jefa de Natalia Guillén» son proposiciones, ya que se puede determinar si son verdaderas o falsas. En el primer caso, la función «es una vendedora de la empresa» tiene un argumento (Paloma Poy) y en el segundo caso, la función «es jefa de» tiene dos argumentos (Paloma Poy y Natalia Guillén).

Si un predicado tiene una variable, como en «`x` es una vendedora de la empresa», esta variable debe tener un *rango* asociado. Cuando la variable se sustituye por alguno de los valores de su rango, la proposición puede ser cierta; para otros valores puede ser falsa. Por ejemplo, si el rango de `x` es el conjunto de

todas las personas y reemplazamos x por Paloma Poy, la proposición «Paloma Poy es una vendedora de la empresa» es cierta. Pero si reemplazamos x por el nombre de una persona que no es vendedora de la empresa, la proposición es falsa.

Si F es un predicado, la siguiente expresión devuelve el conjunto de todos los valores de x para los que F es cierto:

```
x WHERE F(x)
```

Los predicados se pueden conectar mediante AND, OR y NOT para formar *predicados compuestos*.

3.3.1. Cálculo orientado a tuplas

En el cálculo relacional orientado a tuplas, lo que interesa es encontrar tuplas para las que se cumple cierto predicado. El cálculo orientado a tuplas se basa en el uso de *variables tupla*. Una variable tupla es una variable cuyo rango de valores son las tuplas de una relación.

Por ejemplo, para especificar el rango de la variable tupla AX sobre la relación ARTÍCULOS se utiliza la siguiente expresión:

```
RANGE OF AX IS ARTÍCULOS
```

Para expresar la consulta «obtener todas las tuplas AX para las que F(AX) es cierto», se escribe la siguiente expresión:

```
AX WHERE F(AX)
```

donde F es lo que se denomina una *fórmula bien formada*. Por ejemplo, para expresar la consulta «obtener los datos de los artículos con un precio superior a 10€» se puede escribir:

```
RANGE OF AX IS ARTÍCULOS
AX WHERE AX.precio > 10
```

AX.precio se refiere al valor del atributo precio para la tupla AX. Para que se muestren solamente algunos atributos, por ejemplo, codart y descrip, en lugar de todos los atributos de la relación, se deben especificar éstos en la lista de objetivos:

```
RANGE OF AX IS ARTÍCULOS
AX.codart, AX.descrip WHERE AX.precio > 10
```

Hay dos *cuantificadores* que se utilizan en las fórmulas bien formadas para indicar a cuántas instancias se aplica el predicado. El *cuantificador existencial* $\exists$ (*existe*) se utiliza en las fórmulas bien formadas que deben ser ciertas para al menos una instancia.

```
RANGE OF CX IS CLIENTES
∃CX (CX.codcli = FX.codcli AND CX.codpostal = 12003)
```

Esta fórmula bien formada dice que «existe un cliente que tiene el mismo código que el código de cliente de la tupla que ahora se encuentra en la variable de `FACTURAS`, `FX`, y cuyo código postal es 12003». El *cuantificador universal* $\forall$ (*para todo*) se utiliza en las fórmulas bien formadas que deben ser ciertas para todas las instancias.

```
RANGE OF VX IS VENDEDORES
∀VX (VX.codpue ≠ 37758)
```

Esta fórmula bien formada dice que «para todas las tuplas de `VENDEDORES`, la población no es la del código 37758». Utilizando las reglas de las operaciones lógicas, esta fórmula bien formada se puede escribir también del siguiente modo:

```
NOT ∃PX (VX.codpue = 37758)
```

que dice que «no hay ningún vendedor cuya población sea la del código 37758».

Las variables tupla que no están cuantificadas por $\forall$ o $\exists$ se denominan *variables libres*. Si están cuantificadas, se denominan *variables ligadas*. El cálculo, al igual que cualquier lenguaje, tiene una sintaxis que permite construir expresiones válidas. Para que una expresión no sea ambigua y tenga sentido, debe seguir esta sintaxis:

- Si P es un predicado con n argumentos y $t_1, t_2, \ldots, t_n$ son constantes o variables, entonces $P(t_1, t_2, \ldots, t_n)$ es una fórmula bien formada n-ária.
- Si t_1 y t_2 son constantes o variables del mismo dominio y θ es un operador de comparación ($<, \leq, >, \geq, =, \neq$), entonces $t_1 \theta t_2$ es una fórmula bien formada.
- Si P_1 y P_2 son fórmulas bien formadas, también lo son su conjunción P_1 *AND* P_2, su disyunción P_1 *OR* P_2 y la negación *NOT* P_1. Además, si P es una fórmula bien formada que tiene una variable libre X, entonces $\exists X(P)$ y $\forall X(P)$ también son fórmulas bien formadas.

Ejemplo 3.12 *Obtener un listado de los clientes que tienen facturas con descuento.*

Esta petición se puede escribir en términos del cálculo: «un cliente debe salir en el listado si existe alguna tupla en `FACTURAS` que tenga su código de cliente y que tenga descuento (`dto`)».

```
RANGE OF CX IS CLIENTES
RANGE OF FX IS FACTURAS
CX WHERE ∃FX (FX.codcli = CX.codcli AND FX.dto > 0)
```

Nótese que formulando la consulta de este modo no se indica la estrategia a seguir para ejecutarla, por lo que el sistema tiene libertad para decidir qué operaciones hacer y en qué orden. En el álgebra relacional se hubiera formulado así: «Hacer una restricción sobre `FACTURAS` para obtener las tuplas que tienen descuento, y hacer después una concatenación con `CLIENTES`».

Ejemplo 3.13 *Obtener los clientes que tienen descuento en todas sus facturas.*

```
RANGE OF CX IS CLIENTES
RANGE OF FX IS FACTURAS
CX WHERE ∀FX (FX.codcli ≠ CX.codcli OR FX.dto > 0)
```

La expresión anterior es equivalente a esta otra:

```
CX WHERE NOT ∃FX (FX.codcli = CX.codcli AND FX.dto ≤ 0)
```

Y también es equivalente a la siguiente:

```
CX WHERE ∀FX (IF FX.codcli = CX.codcli THEN FX.dto > 0)
```

ya que la expresión `IF p THEN q` es equivalente a la expresión `NOT p OR q`.

3.3.2. Cálculo orientado a dominios

En el cálculo relacional orientado a dominios las variables toman sus valores en dominios, en lugar de tomar valores de tuplas de relaciones. Otra diferencia con el cálculo orientado a tuplas es que en el cálculo orientado a dominios hay un tipo de comparación adicional, a la que se denomina *ser miembro de*. Esta condición tiene la forma:

$\mathtt{R(a_1:v_1,\ a_2:v_2,\ ...)}$

donde los $\mathtt{a}_i$ son atributos de la relación `R` y los $\mathtt{v}_i$ son variables dominio o constantes. La condición se evalúa a verdadero si existe alguna tupla en `R` que tiene los valores especificados en los atributos especificados. Por ejemplo, la siguiente condición:

```
VENDEDORES(codpostal:12003, codjefe:5)
```

se evaluará a verdadero si hay algún empleado con código postal 12003 y cuyo jefe es el vendedor 5. Y la condición:

```
VENDEDORES(codpostal:cpx, codjefe:cjx)
```

será cierta si hay alguna tupla en `VENDEDORES` que tenga en `codpostal` el valor actual de la variable dominio `cpx` y que tenga en `codjefe` el valor actual de la variable dominio `cjx`.

Ejemplo 3.14 *Obtener el nombre de los vendedores cuyo jefe no es el 5, y cuyo código postal es 12003.*

```
nmx WHERE ∃cjx ∃cpx (cjx ≠ 5 AND cpx = 12003
AND VENDEDORES(nombre:nmx, codjefe:cjx, codpostal:cpx))
```

3.4. Otros lenguajes

Aunque el cálculo relacional es difícil de entender y de usar, tiene una propiedad muy atractiva: es un lenguaje no procedural. Esto ha hecho que se busquen técnicas no procedurales algo más sencillas, dando como resultado dos nuevas categorías de lenguajes relacionales: orientados a transformaciones y gráficos.

Los *lenguajes orientados a transformaciones* son lenguajes no procedurales que utilizan relaciones para transformar los datos de entrada en la salida deseada. Estos lenguajes tienen estructuras que son fáciles de utilizar y que permiten expresar lo que se desea en términos de lo que se conoce. Uno de estos lenguajes es SQL (*Structured Query Language*).

Los *lenguajes gráficos* visualizan en pantalla una fila vacía de cada una de las tablas que indica el usuario. El usuario rellena estas filas con un ejemplo de lo que desea y el sistema devuelve los datos que siguen tal ejemplo. Uno de estos lenguajes es QBE (*Query-by-Example*).

Otra categoría son los *lenguajes de cuarta generación (4GL)*, que permiten diseñar una aplicación a medida, utilizando un conjunto limitado de órdenes en un entorno amigable (normalmente un entorno de menús). Algunos sistemas aceptan cierto lenguaje natural, una versión restringida del idioma inglés, al que algunos llaman *lenguaje de quinta generación (5GL)*, aunque todavía se encuentra en desarrollo.

Capítulo 4

Lenguaje SQL

Introducción y objetivos

Las siglas SQL corresponden a *Structured Query Language*, un lenguaje estándar que permite manejar los datos de una base de datos relacional. La mayor parte de los SGBD relacionales implementan este lenguaje y mediante él se realizan todo tipo de accesos a la base de datos. En este capítulo se hace una presentación del lenguaje SQL, haciendo énfasis en la sentencia de consulta de datos, la sentencia `SELECT`.

Al finalizar este capítulo, el estudiante debe ser capaz de:

- Emplear la sentencia `CREATE TABLE` para crear tablas a partir de una especificación dada.
- Emplear las sentencias `INSERT, UPDATE, DELETE` para insertar, actualizar y borrar datos de tablas de una base de datos.
- Emplear la sentencia `SELECT` para responder a cualquier consulta de datos sobre una base de datos dada.
- Especificar una sentencia `SELECT` equivalente a otra dada que no haga uso de los operadores que se indiquen, con el objetivo de intentar acelerar el tiempo de respuesta.

4.1. Bases de datos relacionales

Como se ha visto en capítulos anteriores, una base de datos relacional está formada por un conjunto de relaciones. A las relaciones, en SQL, se las denomina *tablas*. Cada tabla tiene una serie de *columnas* (son los atributos). Cada columna tiene un nombre distinto y es de un tipo de datos (entero, real, carácter, fecha, etc.). En las tablas se insertan *filas* (son las tuplas), que después se pueden consultar, modificar o borrar.

No se debe olvidar que cada tabla tiene una clave primaria, que estará formada por una o varias columnas de esa misma tabla. Sobre las claves primarias se debe hacer respetar una regla de integridad fundamental: la regla de integridad de entidades. La mayoría de los SGBD relacionales se encargan de hacer respetar esta regla automáticamente.

Por otra parte, las relaciones entre los datos de distintas tablas se establecen mediante las claves ajenas. Una clave ajena es una columna o un conjunto de columnas de una tabla que hace referencia a la clave primaria de otra tabla (o de ella misma). Para las claves ajenas también se debe cumplir una regla de integridad fundamental: la regla de integridad referencial. Muchos SGBD relacionales permiten que el usuario establezca las reglas de comportamiento de las claves ajenas que permiten hacer respetar esta regla.

4.2. Descripción de la base de datos

En este apartado se presenta de nuevo la base de datos con la que se ha trabajado en capítulos anteriores y que es la que se utilizará para estudiar el lenguaje SQL en este capítulo. Para evitar problemas de implementación se han omitido las tildes en los nombres de tablas y columnas.

La base de datos está formada por las tablas que aparecen a continuación. Las columnas subrayadas representan la clave primaria de cada tabla.

```
  CLIENTES(codcli, nombre, direccion, codpostal, codpue)
VENDEDORES(codven, nombre, direccion, codpostal, codpue, codjefe)
   PUEBLOS(codpue, nombre, codpro)
PROVINCIAS(codpro, nombre)
 ARTICULOS(codart, descrip, precio, stock, stock_min, dto)
  FACTURAS(codfac, fecha, codcli, codven, iva, dto)
LINEAS_FAC(codfac, linea, cant, codart, precio, dto)
```

A continuación se especifican las claves ajenas y si aceptan nulos:

CLIENTES	$\xrightarrow{\texttt{codpue}}$	PUEBLOS	: No acepta nulos.
VENDEDORES	$\xrightarrow{\texttt{codpue}}$	PUEBLOS	: No acepta nulos.
VENDEDORES	$\xrightarrow{\texttt{codjefe}}$	VENDEDORES	: Acepta nulos.
PUEBLOS	$\xrightarrow{\texttt{codpro}}$	PROVINCIAS	: Acepta nulos.
FACTURAS	$\xrightarrow{\texttt{codcli}}$	CLIENTES	: Acepta nulos.
FACTURAS	$\xrightarrow{\texttt{codven}}$	VENDEDORES	: Acepta nulos.
LINEAS_FAC	$\xrightarrow{\texttt{codfac}}$	FACTURAS	: No acepta nulos.
LINEAS_FAC	$\xrightarrow{\texttt{codart}}$	ARTICULOS	: No acepta nulos.

La información contenida en esta base de datos pertenece a una empresa de venta de artículos eléctricos. A continuación se describe el contenido de cada tabla.

La tabla `PROVINCIAS` almacena información sobre las provincias de España. De cada provincia se almacena su nombre (`nombre`) y un código que la identifica (`codpro`).

La tabla `PUEBLOS` contiene los nombres (`nombre`) de los pueblos de España. Cada pueblo se identifica por un código que es único (`codpue`) y tiene una referencia a la provincia a la que pertenece (`codpro`).

La tabla `CLIENTES` contiene los datos de los clientes: código que identifica a cada uno (`codcli`), nombre y apellidos (`nombre`), calle y número (`direccion`), código postal (`codpostal`) y una referencia a su población (`codpue`).

La tabla `VENDEDORES` contiene los datos de los vendedores de la empresa: código que identifica a cada uno (`codven`), nombre y apellidos (`nombre`), calle y número (`direccion`), código postal (`codpostal`), una referencia a su población (`codpue`) y una referencia al vendedor del que depende (`codjefe`), si es el caso.

En la tabla `ARTICULOS` se tiene el código que identifica a cada artículo (`codart`), su descripción (`descrip`), el precio de venta actual (`precio`), el número de unidades del artículo que hay en el almacén (`stock`), si se conocen, la cantidad mínima que se desea mantener almacenada (`stock_min`), si es que la hay, y si el artículo está en oferta, el descuento (`dto`) que se debe aplicar cuando se venda.

La tabla `FACTURAS` contiene las cabeceras de las facturas correspondientes a las compras realizadas por los clientes. Cada factura tiene un código único (`codfac`), la fecha en que se ha realizado (`fecha`), así como el IVA (`iva`) y el descuento que se le ha aplicado (`dto`). Si el IVA o el descuento no se especifican, se deben interpretar como el valor cero (sin IVA o sin descuento). Es imporante tener en cuenta que se está haciendo un mal uso de los nulos, ya que interpretar los nulos con valores supone un trabajo extra cuando se hacen las consultas. Sin embargo, en muchas bases de datos se hace este uso no apropiado de los nulos y, por lo tanto, el estudio del lenguaje SQL requiere aprender manejarse con ellos. Cada factura también hace referencia al cliente

al que pertenece (`codcli`) y al vendedor que la ha realizado (`codven`). Ambas claves ajenas aceptan nulos.

Las líneas de cada factura se encuentran en la tabla `LINEAS_FAC`, identificándose cada una por el número de línea que ocupa dentro de la factura (`codfac, linea`). En cada una de ellas se especifica la cantidad de unidades (`cant`) del artículo que se compra (`codart`), el precio de venta por unidad (`precio`) y el descuento que se aplica sobre dicho precio (`dto`), si es que el artículo está en promoción. Si el descuento no se especifica, se debe interpretar como sin descuento (valor cero).

La figura 4.1 muestra el esquema de la base de datos gráficamente.

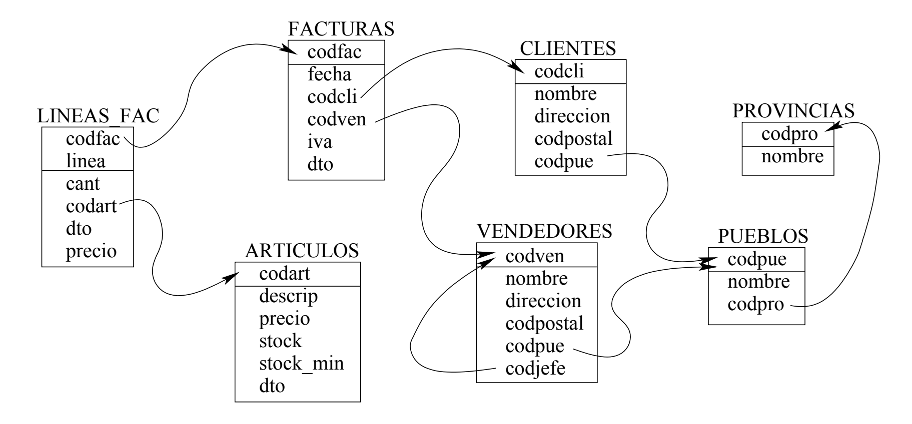


Figura 4.1: Esquema de la base de datos que se utilizará en los ejemplos.

4.3. Visión general del lenguaje

Normalmente, cuando un SGBD relacional implementa el lenguaje SQL, todas las acciones que se pueden llevar a cabo sobre el sistema se realizan mediante sentencias de este lenguaje. Dentro de SQL hay varios tipos de sentencias que se agrupan en tres conjuntos:

- *Sentencias de definición de datos*: son las sentencias que permiten crear tablas, alterar su definición y eliminarlas. En una base de datos relacional existen otros tipos de objetos además de las tablas, como las vistas, los índices y los disparadores, que se estudiarán más adelante. Las sentencias para crear, alterar y eliminar vistas e índices también pertenecen a este conjunto.

- *Sentencias de manejo de datos*: son las sentencias que permiten insertar datos en las tablas, consultarlos, modificarlos y borrarlos.

- *Sentencias de control*: son las sentencias que utilizan los administradores de la base de datos para realizar sus tareas, como por ejemplo crear usuarios y concederles o revocarles privilegios.

Las sentencias de SQL se pueden escribir tanto en mayúsculas como en minúsculas, y lo mismo sucede con los nombres de las tablas y de las columnas. Para facilitar la lectura de los ejemplos, se utilizará mayúsculas para las palabras clave del lenguaje y minúsculas para los nombres de tablas y de columnas. En los ejemplos se introducirán espacios en blanco para tabular las expresiones. Las sentencias de SQL terminan siempre con el carácter punto y coma (;).

4.3.1. Creación de tablas

Para crear una tabla en una base de datos se utiliza la sentencia CREATE TABLE. Su sintaxis es la siguiente:

```
CREATE TABLE nombre_tabla (
 { nombre_columna tipo_datos
                  [ DEFAULT expr ]
                  [ restrición_columna [, ... ] ]
   | restricción_tabla } [, ... ]
);
```

donde restricción_columna es:

```
[ CONSTRAINT nombre_restricción ]
  { NOT NULL | NULL | UNIQUE | PRIMARY KEY | CHECK (expr) |
    REFERENCES tablaref [ ( columnaref ) ]
    [ ON DELETE acción ] [ ON UPDATE acción ] }
  [ DEFERRABLE | NOT DEFERRABLE ]
  [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

y restricción_tabla es:

```
[ CONSTRAINT nombre_restricción ]
  { UNIQUE ( nombre_columna [, ... ] ) |
    PRIMARY KEY ( nombre_columna [, ... ] ) |
    CHECK ( expr ) |
    FOREIGN KEY ( nombre_columna [, ... ] )
        REFERENCES tablaref [ ( columnaref [, ... ] ) ]
        [ MATCH FULL | MATCH PARTIAL ]
        [ ON DELETE acción ] [ ON UPDATE acción ] }
```

A continuación se especifica el significado de cada identificador y de cada cláusula de la sentencia CREATE TABLE:

- nombre_tabla: nombre de la nueva tabla.
- nombre_columna: nombre de una columna de la tabla.

- `tipo_datos`: tipo de datos de la columna.
- `DEFAULT expr`: asigna un valor por defecto a la columna junto a la que aparece; este valor se utilizará cuando en una inserción no se especifique valor para la columna.
- `CONSTRAINT nombre_restricción`: a las restricciones que se definen sobre columnas y sobre tablas se les puede dar un nombre (si no se hace, el sistema generará un nombre automáticamente).
- `NOT NULL`: la columna no admite nulos.
- `NULL`: la columna admite nulos (se toma por defecto si no se especifica `NOT NULL`).
- `UNIQUE` especificada como restricción de columna indica que la columna sólo puede contener valores únicos. `UNIQUE (nombre_columna [,...])` especificada como restricción de tabla indica que el grupo de columnas sólo pueden contener grupos de valores únicos. Mediante esta cláusula se especifican las claves alternativas.
- `PRIMARY KEY` especificada como restricción de columna o bien `PRIMARY KEY (nombre_columna[,...])` especificada como restricción de tabla indica la columna o el grupo de columnas que forman la clave primaria de la tabla. Los valores de la clave primaria, además de ser únicos, deberán ser no nulos.
- `CHECK (expr)`: permite incluir reglas de integridad específicas que se comprueban para cada fila que se inserta o que se actualiza. La expresión es un predicado que produce un resultado booleano. Si se especifica a nivel de columna, en la expresión sólo puede hacerse referencia a esta columna. Si se especifica a nivel de tabla, en la expresión puede hacerse referencia a varias columnas. Por ahora no se puede incluir subconsultas en esta cláusula.
- Restricción de columna:
  ```
  REFERENCES tablaref [ ( columnaref ) ]
  [ ON DELETE acción ] [ ON UPDATE acción ]
  ```
  Restricción de tabla:
  ```
  FOREIGN KEY ( nombre_columna [, ... ] )
  REFERENCES tablaref [ ( columnaref [, ... ] ) ]
  [ MATCH FULL | MATCH PARTIAL ]
  [ ON DELETE acción ] [ ON UPDATE acción ]
  ```

  La restricción de columna `REFERENCES` permite indicar que la columna hace referencia a una columna de otra tabla. Si la referencia apunta a la clave primaria, no es necesario especificar el nombre de la columna a

la que se hace referencia (estamos definiendo una clave ajena). Cuando se añade o actualiza un valor en esta columna, se comprueba que dicho valor existe en la tabla referenciada. Cuando la restricción es a nivel de tabla (`FOREIGN KEY`) hay dos tipos de comprobación: `MATCH FULL` y `MATCH PARTIAL`. Con `MATCH FULL`, si la clave ajena está formada por varias columnas y admite nulos, esta comprobación es la que corresponde a la regla de integridad referencial: en cada fila, o todas las columnas de la clave ajena tienen valor o ninguna de ellas lo tiene (todas son nulas), pero no se permite que en una misma fila, algunas sean nulas y otras no. Con `MATCH PARTIAL`, si la clave ajena está formada por varias columnas y admite nulos, se permiten claves ajenas parcialmente nulas y se comprueba que en la tabla referenciada se podría apuntar a alguna de sus filas si los nulos se sustituyeran por los valores adecuados.

Además, se pueden establecer reglas de comportamiento para cada clave ajena cuando se borra o se actualiza el valor referenciado. En ambos casos hay cuatro posibles opciones que se enumeran a continuación. `NO ACTION` produce un error por intento de violación de una restricción. `RESTRICT` es igual que `NO ACTION`. `CASCADE` borra/actualiza las filas que hacen referencia al valor borrado/actualizado. `SET NULL` pone un nulo en las filas donde se hacía referencia al valor borrado/actualizado. `SET DEFAULT` pone el valor por defecto en las filas donde se hacía referencia al valor borrado/actualizado.

A continuación se muestra la sentencia de creación de la tabla `LINEAS_FAC`:

```
CREATE TABLE lineas_fac (
    codfac   NUMERIC(6,0) NOT NULL,
    linea    NUMERIC(2,0) NOT NULL,
    cant     NUMERIC(5,0) NOT NULL,
    codart   VARCHAR(8)   NOT NULL,
    precio   NUMERIC(6,2) NOT NULL,
    dto      NUMERIC(2,0),
    CONSTRAINT cp_lineas_fac PRIMARY KEY (codfac, linea),
    CONSTRAINT ca_lin_fac FOREIGN KEY (codfac)
               REFERENCES facturas(codfac)
               ON UPDATE CASCADE ON DELETE CASCADE,
    CONSTRAINT ca_lin_art FOREIGN KEY (codart)
               REFERENCES articulos(codart)
               ON UPDATE CASCADE ON DELETE RESTRICT,
    CONSTRAINT ri_dto_lin CHECK (dto BETWEEN 0 AND 50)
);
```

4.3.2. Inserción de datos

Una vez creada una tabla podemos introducir datos en ella mediante la sentencia `INSERT`, como se muestra en los siguientes ejemplos:

```
INSERT INTO facturas(codfac,fecha,       codcli,codven,iva,dto )
             VALUES(6600,  '30/04/2007',111,   55,    0,  NULL);
INSERT INTO lineas_fac(codfac,linea,cant,codart,  precio,dto)
               VALUES(6600,  1,    4,   'L76425',3.16,  25 );
INSERT INTO lineas_fac(codfac,linea,cant,codart,  precio,dto)
               VALUES(6600,  2,    5,   'B14017',2.44,  25 );
INSERT INTO lineas_fac(codfac,linea,cant,codart,  precio,dto)
               VALUES(6600,  3,    7,   'L92117',4.39,  25 );
```

Mediante estas sentencias se ha introducido la cabecera de una factura y tres de sus líneas. Nótese que tanto las cadenas de caracteres como las fechas, se introducen entre comillas simples. Para introducir nulos se utiliza la expresión `NULL`.

Algunos SGBD relacionales permiten insertar varias filas en una misma tabla mediante una sola sentencia `INSERT`, y realizan las inserciones de un modo más eficiente que si se hace mediante varias sentencias independientes. Así, la tres inserciones que se han realizado en la tabla `LINEAS_FAC` también se pueden realizar mediante la siguiente sentencia:

```
INSERT INTO lineas_fac(codfac,linea,cant,codart,  precio,dto)
               VALUES(6600,  1,    4,   'L76425',3.16,  25 ),
                     (6600,  2,    5,   'B14017',2.44,  25 ),
                     (6600,  3,    7,   'L92117',4.39,  25 );
```

4.3.3. Consulta de datos

Una vez se ha visto cómo almacenar datos en la base de datos, interesa conocer cómo se puede acceder a dichos datos para consultarlos. Para ello se utiliza la sentencia `SELECT`. Por ejemplo:

```
SELECT *
FROM   facturas;
```

En primer lugar aparece la palabra `SELECT`, que indica que se va a realizar una consulta. A continuación, el `*` indica que se desea ver el contenido de todas las columnas de la tabla consultada. El nombre de esta tabla es el que aparece tras la palabra `FROM`, en este caso, la tabla `facturas`.

Esta sentencia es, sin lugar a dudas, la más compleja del lenguaje de manejo de datos y es por ello que gran parte de este capítulo se centra en su estudio.

4.3.4. Actualización y eliminación de datos

Una vez insertados los datos es posible actualizarlos o eliminarlos mediante las sentencias UPDATE y DELETE, respectivamente. Para comprender el funcionamiento de estas dos sentencias es imprescindible conocer bien el funcionamiento de la sentencia SELECT. Esto es así porque para poder actualizar o eliminar datos que se han almacenado es preciso encontrarlos antes. Y por lo tanto, la cláusula de estas sentencias que establece las condiciones de búsqueda de dichos datos (WHERE) se especifica del mismo modo que las condiciones de búsqueda cuando se hace una consulta.

Sin embargo, antes de pasar al estudio de la sentencia SELECT se muestran algunos ejemplos de estas dos sentencias.

```
UPDATE facturas        UPDATE facturas
SET    dto = 0         SET    codven = 105
WHERE  dto IS NULL;    WHERE  codven IN ( SELECT codven
                                          FROM   vendedores
                                          WHERE  codjefe = 105 );

DELETE FROM facturas   DELETE FROM facturas
WHERE  codcli = 333;   WHERE  iva = ( SELECT MIN(iva)
                                      FROM   facturas );
```

4.4. Estructura básica de la sentencia SELECT

La sentencia SELECT consta de varias cláusulas. A continuación se muestran algunas de ellas:

```
SELECT [ DISTINCT ] { * | columna [ , columna ] }
FROM   tabla
[ WHERE condición_de_búsqueda ]
[ ORDER BY columna [ ASC | DESC ]
           [,columna [ ASC | DESC ] ];
```

El orden en que se tienen en cuenta las distintas cláusulas durante la ejecución y la función de cada una de ellas es la siguiente:

- FROM: especifica la tabla sobre la que se va a realizar la consulta.
- WHERE: si sólo se debe mostrar un subconjunto de las filas de la tabla, aquí se especifica la condición que deben cumplir las filas a mostrar; esta condición será un predicado booleano con comparaciones unidas por AND/OR.
- SELECT: aquí se especifican las columnas a mostrar en el resultado; para mostrar todas las columnas se utiliza *.

- DISTINCT: es un modificador que se utiliza tras la cláusula SELECT para que no se muestren filas repetidas en el resultado (esto puede ocurrir sólo cuando en la cláusula SELECT se prescinde de la clave primaria de la tabla o de parte de ella, si es compuesta).
- ORDER BY: se utiliza para ordenar el resultado de la consulta.

La cláusula ORDER BY, si se incluye, es siempre la última en la sentencia SELECT. La ordenación puede ser ascendente o descendente y puede basarse en una sola columna o en varias.

La sentencia del siguiente ejemplo muestra los datos de todos los clientes, ordenados por el código postal, descendentemente. Además, todos los clientes de un mismo código postal aparecerán ordenados por el nombre, ascendentemente.

```
SELECT *
FROM   clientes
ORDER BY codpostal DESC, nombre;
```

4.4.1. Expresiones en SELECT y WHERE

En las cláusulas SELECT y WHERE, además de columnas, también se pueden incluir expresiones que contengan columnas y constantes, así como funciones. Las columnas y expresiones especificadas en la cláusula SELECT se pueden renombrar al mostrarlas en el resultado mediante AS.

Si el resultado de una consulta se debe mostrar ordenado según el valor de una expresión de la cláusula SELECT, esta expresión se indica en la cláusula ORDER BY mediante el número de orden que ocupa en la cláusula SELECT.

```
SELECT precio, ROUND(precio * 0.8, 2) AS rebajado
FROM   articulos
ORDER BY 2;
```

4.4.2. Nulos

Cuando no se ha insertado un valor en una columna de una fila se dice que ésta es nula. Un nulo no es un valor: un nulo implica ausencia de valor. Para saber si una columna es nula se debe utilizar el operador de comparación IS NULL y para saber si no es nula, el operador es IS NOT NULL.

Cuando se realiza una consulta de datos, los nulos se pueden interpretar como valores mediante la función COALESCE(columna, valor_si_nulo). Esta función devuelve valor_si_nulo en las filas donde columna es nula; si no, devuelve el valor de columna.

```
SELECT codfac, fecha, codcli, COALESCE(iva, 0) AS iva,
       iva AS iva_null, COALESCE(dto, 0) AS dto
FROM   facturas
WHERE  codcli < 50
AND    (iva = 0 OR iva IS NULL);
```

La condición `(iva=0 OR iva IS NULL)` es equivalente a `COALESCE(iva,0)=0`.

4.4.3. Tipos de datos

Los tipos de datos disponibles se deben consultar en el manual del SGBD relacional que se esté utilizando. Puesto que las prácticas de las asignaturas para las que se edita este libro se realizan bajo PostgreSQL, se presentan aquí los tipos de datos que se han usado en este SGBD para crear las tablas. Todos ellos pertenecen al estándar de SQL.

- `VARCHAR(n)`: Cadena de hasta `n` caracteres.
- `NUMERIC(n,m)`: Número con `n` dígitos, de los cuales `m` se encuentran a la derecha del punto decimal.
- `DATE`: Fecha formada por día, mes y año. Para guardar fecha y hora se debe utilizar el tipo `TIMESTAMP`.
- `BOOLEAN`: Aunque este tipo no se ha utilizado en la base de datos de prácticas, es interesante conocer su existencia. El valor verdadero se representa mediante `TRUE` y el falso mediante `FALSE`. Cuando se imprimen estos valores, se muestra el carácter `'t'` para verdadero y el carácter `'f'` para falso.

Hay que tener siempre en cuenta que el nulo no es un valor, sino que implica ausencia de valor. El nulo se representa mediante `NULL` y cuando se imprime no se muestra nada.

4.5. Funciones y operadores

4.5.1. Operadores lógicos

Los operadores lógicos son `AND`, `OR` y `NOT`. SQL utiliza una lógica booleana de tres valores y la evaluación de las expresiones con estos operadores es la que se muestra en la siguiente tabla:

a	b	a `AND` b	a `OR` b	`NOT` b
True	True	True	True	False
True	False	False	True	True
True	Null	Null	True	Null
False	False	False	False	
False	Null	False	Null	
Null	Null	Null	Null	

4.5.2. Operadores de comparación

`<`	Menor que.
`>`	Mayor que.
`<=`	Menor o igual que.
`>=`	Mayor o igual que.
`=`	Igual que.
`<>  !=`	Distinto de.
`a BETWEEN x AND y`	Equivale a: `a >= x AND a <= y`
`a NOT BETWEEN x AND y`	Equivale a: `a < x OR a > y`
`a IS NULL`	Devuelve True si `a` es nulo.
`a IS NOT NULL`	Devuelve True si `a` es no nulo.
`a IN (v1, v2, ...)`	Equivale a: `a = v1 OR a = v2 OR ...`

4.5.3. Operadores matemáticos

`+`	Suma.
`-`	Resta.
`*`	Multiplicación.
`/`	División (si es entre enteros, trunca el resultado).
`%`	Resto de la división entera.
`^`	Potencia (3^2 = 9).
`|/`	Raíz cuadrada (|/25 = 5).
`||/`	Raíz cúbica (||/27 = 3).
`!`	Factorial (5! = 120).
`!!`	Factorial como operador prefijo (!!5 = 120).
`@`	Valor absoluto.

No se han incluido en esta lista los operadores que realizan operaciones sobre tipos de datos binarios.

4.5.4. Funciones matemáticas

`ABS(x)`	Valor absoluto de `x`.
`SIGN(x)`	Devuelve el signo de `x` (-1, 0, 1).
`MOD(x,y)`	Resto de la división entera de `x` entre `y`.
`SQRT(x)`	Raíz cuadrada de `x`.
`CBRT(x)`	Raíz cúbica de `x`.
`CEIL(x)`	Entero más cercano por debajo de `x`.
`FLOOR(x)`	Entero más cercano por encima de `x`.
`ROUND(x)`	Redondea al entero más cercano.
`ROUND(x,n)`	Redondea `x` a `n` dígitos decimales, si `n` es positivo. Si `n` es negativo, redondea al entero más cercano a `x` múltiplo de 10^n.
`TRUNC(x)`	Trunca `x`.
`TRUNC(x,n)`	Trunca `x` a `n` dígitos decimales, si `n` es positivo. Si `n` es negativo, trunca al entero más cercano por debajo de `x` múltiplo de 10^n.

Además de éstas, se suelen incluir otras muchas funciones para: calcular logaritmos, convertir entre grados y radianes, funciones trigonométricas, etc. Se aconseja consultar los manuales del SGBD que se esté utilizando, para conocer las funciones que se pueden utilizar y cuál es su sintaxis.

4.5.5. Operadores y funciones de cadenas de caracteres

En SQL, las cadenas de caracteres se delimitan por comillas simples: `'abc'`. Los operadores y funciones para trabajar con cadenas son los siguientes:

`cadena || cadena`	Concatena dos cadenas.
`cadena LIKE expr`	Devuelve `TRUE` si la cadena sigue el patrón de la cadena que se pasa en `expr`. En `expr` se pueden utilizar comodines: _ para un solo carácter y % para cero o varios caracteres.
`LENGTH(cadena)`	Número de caracteres que tiene la cadena.
`CHAR_LENGTH(cadena)`	Es la función del estándar equivalente a `LENGTH`.
`POSITION(subcadena IN cadena)`	Posición de inicio de la subcadena en la cadena.
`SUBSTR(cadena, n [, long])`	Devuelve la subcadena de la cadena que empieza en la posición `n` (`long` fija el tamaño máximo de la subcadena; si no se especifica, devuelve hasta el final).

`SUBSTRING(cadena FROM n [FOR long])`
Es la función del estándar equivalente a `SUBSTR`: devuelve la subcadena de la cadena que empieza en la posición `n` (`long` fija el tamaño máximo de la subcadena; si no se especifica, devuelve hasta el final).

`LOWER(cadena)` Devuelve la cadena en minúsculas.

`UPPER(cadena)` Devuelve la cadena en mayúsculas.

`BTRIM(cadena)` Elimina los espacios que aparecen por delante y por detrás en la cadena.

`LTRIM(cadena)` Elimina los espacios que aparecen por delante (izquierda) en la cadena.

`RTRIM(cadena)` Elimina los espacios que aparecen por detrás (derecha) de la cadena.

`BTRIM(cadena, lista)` Elimina en la cadena la subcadena formada sólo por caracteres que aparecen en la lista, tanto por delante como por detrás.

`LTRIM(cadena, lista)` Funciona como `BTRIM` pero sólo por delante (izquierda).

`RTRIM(cadena, lista)` Funciona como `BTRIM` pero sólo por detrás (derecha).

`TRIM(lado lista FROM cadena)`
Es la función del estándar equivalente a `BTRIM` si lado es `BOTH`, equivalente a `LTRIM` si lado es `LEADING` y equivalente a `RTRIM` si lado es `TRAILING`.

`CHR(n)` Devuelve el carácter cuyo código ASCII viene dado por `n`.

`INITCAP(cadena)` Devuelve la cadena con la primera letra de cada palabra en mayúscula y el resto en minúsculas.

`LPAD(cadena, n, [, c])` Devuelve la cadena rellenada por la izquierda con el carácter `c` hasta completar la longitud especificada por `n` (si no se especifica `c`, se rellena de espacios). Si la longitud de la cadena es de más de `n` caracteres, se trunca por el final.

`RPAD(cadena, n, [, c])` Devuelve la cadena rellenada por la derecha con el carácter `c` hasta completar la longitud especificada por `n` (si no se especifica `c`, se rellena de espacios). Si la longitud de la cadena es de más de `n` caracteres, se trunca por el final.

4.5.6. Operadores y funciones de fecha

El tipo de datos DATE tiene operadores y funciones, como el resto de tipos.[1] En este apartado se muestran aquellos más utilizados, pero se remite al lector a los manuales del SGBD que esté utilizando para conocer el resto.

En primer lugar se verán las funciones que permiten convertir entre distintos tipos de datos. Todas ellas tienen la misma estructura: se les pasa un `dato` de un tipo, que se ha de convertir a otro tipo según el patrón indicado mediante un `formato`.

`TO_CHAR(dato, formato)`	Convierte el `dato` de cualquier tipo a cadena de caracteres.
`TO_DATE(dato, formato)`	Convierte el `dato` de tipo cadena a fecha.
`TO_NUMBER(dato, formato)`	Convierte el `dato` de tipo cadena a número.

A continuación se muestran algunos de los patrones que se pueden especificar en los formatos:

Conversiones fecha/hora:

`HH`	Hora del día (1:12).
`HH12`	Hora del día (1:12).
`HH24`	Hora del día (1:24).
`MI`	Minuto (00:59).
`SS`	Segundo (00:59).
`YYYY`	Año.
`YYY`	Últimos tres dígitos del año.
`YY`	Últimos dos dígitos del año.
`Y`	Último dígito del año.
`MONTH`	Nombre del mes.
`MON`	Nombre del mes abreviado.
`DAY`	Nombre del día.
`DY`	Nombre del día abreviado.
`DDD`	Número del día dentro del año (001:366).
`DD`	Número del día dentro del mes (01:31).
`D`	Número del día dentro de la semana (1:7 empezando en domingo).
`WW`	Número de la semana en el año (1:53).
`W`	Número de la semana en el mes (1:5).
`Q`	Número del trimestre (1:4).

[1] En PostgreSQL se puede escoger el modo de visualizar las fechas mediante `SET DATESTYLE`. Para visualizar las fechas con formato `día/mes/año` se debe ejecutar la orden `SET DATESTYLE TO EUROPEAN, SQL;`

Conversiones numéricas:

9	Dígito numérico.
S	Valor negativo con signo menos.
.	Punto decimal.
,	Separador de miles.

Cuando el formato muestra un nombre, utilizando en el patrón de forma adecuada las mayúsculas y minúsculas, se cambia el modo en que se muestra la salida. Por ejemplo, `MONTH` muestra el nombre del mes en mayúsculas, `Month` lo muestra sólo con la inicial en mayúscula y `month` lo muestra todo en minúsculas. Cualquier carácter que se especifique en el formato y que no coincida con ningún patrón, se copia en la salida del mismo modo en que está escrito. A continuación se muestran algunos ejemplos:

```
SELECT TO_CHAR(CURRENT_TIMESTAMP, 'HH12 horas MI m. SS seg.');
SELECT TO_CHAR(CURRENT_DATE, 'Day, dd of month, yyyy');
SELECT TO_NUMBER('-12,454.8','S99,999.9');
```

Las funciones de fecha más habituales son las siguientes:

`CURRENT_DATE`	Función del estándar que devuelve la fecha actual (el resultado es de tipo `DATE`).
`CURRENT_TIME`	Función del estándar que devuelve la hora actual (el resultado es de tipo `TIME`).
`CURRENT_TIMESTAMP`	Función del estándar que devuelve la fecha y hora actuales (el resultado es de tipo `TIMESTAMP`).
`EXTRACT(campo FROM dato)`	Función del estándar que devuelve la parte del `dato` (fecha u hora) indicada por `campo`. El resultado es de tipo `DOUBLE PRECISION`. En `campo` se pueden especificar las siguientes partes: `day`: día del mes (1:31) `dow`: día de la semana (0:6 empezando en domingo) `doy`: día del año (1:366) `week`: semana del año `month`: mes del año (1:12) `quarter`: trimestre del año (1:4) `year`: año `hour`: hora `minute`: minutos `second`: segundos

A continuación se muestran algunos ejemplos de uso de estas funciones:

```
SELECT CURRENT_TIMESTAMP;
SELECT 365 - EXTRACT(DOY FROM CURRENT_DATE) AS dias_faltan;
SELECT EXTRACT(WEEK FROM TO_DATE('24/09/2008','dd/mm/yyyy'));
```

Para sumar o restar días a una fecha se utilizan los operadores `+` y `-`. Por ejemplo, para sumar siete días a la fecha actual se escribe: `CURRENT_DATE+7`.

4.5.7. Función CASE

Los lenguajes de programación procedurales suelen tener sentencias condicionales: si una condición es cierta entonces se realiza una acción, en caso contrario se realiza otra acción distinta. SQL no es un lenguaje procedural; sin embargo, permite un control condicional sobre los datos devueltos en una consulta, mediante la función `CASE`.

A continuación se muestra un ejemplo que servirá para explicar el modo de uso de esta función:

```
SELECT codart, precio,
       CASE WHEN stock > 500 THEN precio*0.8
            WHEN stock BETWEEN 200 AND 500 THEN precio*0.9
            ELSE precio
       END AS precio_con_descuento
FROM   articulos;
```

Esta sentencia muestra, para cada artículo, su código, su precio y un precio con descuento que se obtiene en función de su stock: si el stock es superior a 500 unidades, el descuento es del 20 % (se multiplica el precio por 0.8), si el stock está entre las 200 y las 500 unidades, el descuento es del 10 % (se multiplica el precio por 0.9) y si no, el precio se mantiene sin descuento. La columna con el precio de descuento se renombra (`precio_con_descuento`). La función `CASE` termina con `END` y puede tener tantas cláusulas `WHEN` ... `THEN` como se precise.

4.5.8. Funciones COALESCE y NULLIF

La función `COALESCE` devuelve el primero de sus parámetros que es no nulo. La función `NULLIF` devuelve un nulo si `valor1` y `valor2` son iguales; si no, devuelve `valor1`. La sintaxis de estas funciones es la siguiente:

```
COALESCE( valor [, ...] )
NULLIF( valor1, valor2 )
```

Ambas funciones se transforman internamente en expresiones equivalentes con la función `CASE`.

Por ejemplo, la siguiente sentencia:

```
SELECT codart, descrip,
       COALESCE(stock, stock_min, -1)
FROM   articulos;
```

es equivalente a esta otra:

```
SELECT codart, descrip,
       CASE WHEN stock IS NOT NULL THEN stock
            WHEN stock_min IS NOT NULL THEN stock_min
            ELSE -1 END
FROM   articulos;
```

Del mismo modo, la siguiente sentencia:

```
SELECT codart, descrip,
       NULLIF(stock, stock_min)
FROM   articulos;
```

es equivalente a esta otra:

```
SELECT codart, descrip,
       CASE WHEN stock=stock_min THEN NULL
            ELSE stock END
FROM   articulos;
```

Hay que tener siempre mucha precaución con las columnas que aceptan nulos y tratarlos adecuadamente cuando se deba hacer alguna restricción (`WHERE`) sobre dicha columna.

4.5.9. Ejemplos

Ejemplo 4.1 *Se quiere obtener un listado con el código y la fecha de las facturas del año pasado que pertenecen a clientes cuyo código está entre el 50 y el 80. El resultado debe aparecer ordenado por la fecha, descendentemente.*

Al consultar la descripción de la tabla de `FACTURAS` puede verse que la columna fecha es de tipo `DATE`. Por lo tanto, para obtener las facturas del año pasado se debe obtener el año en curso (`CURRENT_DATE`) y quedarse con aquellas cuyo año es una unidad menor. El año de una fecha se obtiene utilizando la función `EXTRACT` tal y como se muestra a continuación.

```
SELECT codfac, fecha
FROM   facturas
WHERE  EXTRACT(year FROM fecha) =
       EXTRACT(year FROM CURRENT_DATE)-1
AND    codcli BETWEEN 50 AND 80
ORDER BY fecha DESC;
```

Ejemplo 4.2 *Mostrar la fecha actual en palabras.*

```
SELECT TO_CHAR(CURRENT_DATE,'Day, dd of month of yyyy');
```

Al ejecutar esta sentencia se observa que quedan huecos demasiado grandes entre algunas palabras:

```
Sunday   , 20 of july      of 2008
```

Esto es así porque para la palabra del día de la semana y la palabra del mes se está dejando el espacio necesario para mostrar la palabra más larga que puede ir en ese lugar. Si se desea eliminar los blancos innecesarios se debe hacer uso de la función `RTRIM`.

```
SELECT RTRIM(TO_CHAR(CURRENT_DATE, 'Day')) ||
       RTRIM(TO_CHAR(CURRENT_DATE, ', dd of month')) ||
       TO_CHAR(CURRENT_DATE, ' of yyyy');

Sunday, 20 of july of 2008
```

Ejemplo 4.3 *Se quiere obtener un listado con los códigos de los vendedores que han hecho ventas al cliente cuyo código es el 54.*

La información que se solicita se extrae de la tabla de `FACTURAS`: el código de vendedor de las facturas de dicho cliente. Puesto que el cliente puede tener varias facturas con el mismo vendedor (`codven` no es clave primaria ni clave alternativa en esta tabla), se debe utilizar el modificador `DISTINCT`.

```
SELECT DISTINCT codven
FROM   facturas
WHERE  codcli = 54;
```

Es muy importante saber de antemano cuándo se debe utilizar el modificador `DISTINCT`.

4.6. Operaciones sobre conjuntos de filas

En el apartado anterior se han presentado algunos de los operadores y de las funciones que se pueden utilizar en las cláusulas `SELECT` y `WHERE` de la sentencia `SELECT`. Mediante estos operadores y funciones construimos expresiones *a nivel de fila*. Por ejemplo, en la siguiente sentencia:

```
SELECT DISTINCT EXTRACT(month FROM fecha) AS meses
FROM   facturas
WHERE  codcli IN (45, 54, 87, 102)
AND    EXTRACT(year FROM fecha) =
       EXTRACT(year FROM CURRENT_DATE)-1;
```

se parte de la tabla `FACTURAS` y se seleccionan las filas que cumplen la condición de la cláusula `WHERE`. A continuación, se toma el valor de la fecha de cada fila seleccionada, se extrae el mes y se muestra éste sin repeticiones.

En este apartado se muestra cómo se pueden realizar operaciones *a nivel de columna*, teniendo en cuenta todas las filas de una tabla (sin cláusula `WHERE`) o bien teniendo en cuenta sólo algunas de ellas (con cláusula `WHERE`). Además, se muestra cómo las funciones de columna se pueden aplicar sobre grupos de filas cuando se hace uso de la cláusula `GROUP BY`. Este uso se hace necesario cuando los cálculos a realizar no son sobre todas las filas de una tabla o sobre un subconjunto, sino que se deben realizar repetidamente para distintos grupos de filas.

4.6.1. Funciones de columna

En ocasiones es necesario contar datos: ¿cuántos clientes hay en Castellón? O también hacer cálculos sobre ellos: ¿a cuánto asciende el IVA cobrado en la factura 3752? SQL proporciona una serie de funciones que se pueden utilizar en la cláusula SELECT y que actúan sobre los valores de las columnas para realizar diversas operaciones como, por ejemplo, sumarlos u obtener el valor máximo o el valor medio, entre otros. Las funciones de columna más habituales son las que se muestran a continuación:

COUNT(*)	Cuenta filas.
COUNT(columna)	Cuenta valores no nulos.
SUM(columna)	Suma los valores de la columna.
MAX(columna)	Obtiene el valor máximo de la columna.
MIN(columna)	Obtiene el valor mínimo de la columna.
AVG(columna)	Obtiene la media de los valores de la columna.

Si no se realiza ninguna restricción en la cláusula WHERE de una sentencia SELECT que utiliza funciones de columna, éstas se aplican sobre todas las filas de la tabla especificada en la cláusula FROM. Sin embargo, cuando se realiza una restricción mediante WHERE, las funciones se aplican sólo sobre las filas que la restricción ha seleccionado.

A continuación, se muestran algunos ejemplos:

```
-- cantidad media por línea de factura
SELECT AVG(cant)
FROM   lineas_fac;

-- cantidad media por línea de factura del artículo
SELECT AVG(cant)
FROM   lineas_fac
WHERE  codart = 'TLFXK2';

-- se puede hacer varios cálculos a la vez
SELECT SUM(cant) AS suma, COUNT(*) AS lineas
FROM   lineas_fac;
```

La función COUNT( ) realiza operaciones distintas dependiendo de su argumento:

COUNT(*)	Cuenta filas.
COUNT(columna)	Cuenta el número de valores no nulos en la columna.
COUNT(DISTINCT columna)	Cuenta el número de valores distintos y no nulos en la columna.

A continuación, se muestra su uso mediante un ejemplo. Se ha creado una tabla P que contiene los datos de una serie de piezas:

```
SELECT * FROM P;

pnum |  pnombre   |   color    | peso |   ciudad
------+------------+------------+------+------------
P1   | tuerca     | verde      |   12 | París
P2   | perno      | rojo       |      | Londres
P3   | birlo      | azul       |   17 | Roma
P4   | birlo      | rojo       |   14 | Londres
P5   | leva       |            |   12 | París
P6   | engrane    | rojo       |   19 | París
```

y se ha ejecutado la siguiente sentencia:

```
SELECT COUNT(*) AS cuenta1, COUNT(color) AS cuenta2,
       COUNT(DISTINCT color) AS cuenta3
FROM   P;
```

El resultado de ejecutarla será el siguiente:

```
 cuenta1 | cuenta2 | cuenta3
---------+---------+---------
       6 |       5 |       3
```

A la vista de los resultados se puede decir que `cuenta1` contiene el número de piezas, `cuenta2` contiene el número de piezas con color y `cuenta3` contiene el número de colores de los que hay piezas.

Las funciones de columna (`SUM, MAX, MIN, AVG`) ignoran los nulos, es decir, los nulos no son tenidos en cuenta en los cálculos. Según esto, se plantea la siguiente pregunta: ¿coincidirá siempre el valor de `media1` y `media2` al ejecutar la siguiente sentencia?

```
SELECT AVG(dto) AS media1, SUM(dto)/COUNT(*) AS media2
FROM   lineas_fac;
```

La respuesta es negativa, ya que en `media1` se devuelve el valor medio de los descuentos no nulos, mientras que en `media2` lo que se devuelve es el valor medio de los descuentos (interpretándose los descuentos nulos como el descuento cero).

Como se ha visto, la función `AVG` calcula la media de los valores no nulos de una columna. Si la tabla de la cláusula `FROM` es la de artículos, la media es por artículo; si la tabla de la cláusula `FROM` es la de facturas, la media es por factura. Cuando se quiere calcular otro tipo de media se debe hacer el cálculo mediante un cociente. Por ejemplo, el número medio de facturas *por mes* durante el año pasado se obtiene dividiendo el número de facturas del año pasado entre doce meses:

```
SELECT COUNT(*)/12 AS media_mensual
FROM   facturas
WHERE  EXTRACT(year FROM fecha) =
       EXTRACT(year FROM CURRENT_DATE)-1;
```

Es importante tener en cuenta que la función COUNT devuelve un entero y que las operaciones entre enteros devuelven resultados enteros. Es decir, la operación SELECT 2/4; devuelve el resultado cero. Por lo tanto, es conveniente multiplicar uno de los operandos por 1.0 para asegurarse de que se opera con números reales. En este caso, será necesario redondear los decimales del resultado a lo que sea preciso:

```
SELECT ROUND(COUNT(*)*1.0/12,2) AS media_mensual
FROM   facturas
WHERE  EXTRACT(year FROM fecha) =
       EXTRACT(year FROM CURRENT_DATE)-1;
```

4.6.2. Cláusula GROUP BY

La cláusula GROUP BY *forma grupos* con las filas que tienen en común los valores de una o varias columnas. Sobre cada grupo se pueden aplicar las funciones de columna que se han estado utilizando hasta ahora (SUM, MAX, MIN, AVG, COUNT), que pasan a denominarse *funciones de grupo*. Estas funciones, utilizadas en la cláusula SELECT, se aplican una vez para cada grupo.

La siguiente sentencia cuenta cuántas facturas tiene cada cliente el año pasado:

```
SELECT codcli, COUNT(*)
FROM   facturas
WHERE  EXTRACT(year FROM fecha) =
       EXTRACT(year FROM CURENT_DATE)-1
GROUP  BY codcli;
```

El modo en que se ejecuta la sentencia se explica a continuación. Se toma la tabla de facturas (FROM) y se seleccionan las filas que cumplen la restricción (WHERE). A continuación, las facturas se separan en grupos, de modo que en un mismo grupo sólo hay facturas de un mismo cliente (GROUP BY codcli), con lo cual hay tantos grupos como clientes hay con facturas del año pasado. Finalmente, de cada grupo se muestra el código del cliente y el número de facturas que hay en el grupo (son las facturas de ese cliente): COUNT(*).

4.6.3. Cláusula HAVING

En la cláusula HAVING, que puede aparecer tras GROUP BY, se utilizan las funciones de grupo para hacer restricciones sobre los grupos que se han formado. La sintaxis de la sentencia SELECT, tal y como se ha visto hasta el momento, es la siguiente:

```
SELECT  [ DISTINCT ] { * | columna [ , columna ] }
FROM    tabla
[ WHERE condición_de_búsqueda ]
[ GROUP BY columna [, columna ]
[ HAVING condición_para_el_grupo ] ]
[ ORDER BY columna [ ASC | DESC ]
          [,columna [ ASC | DESC ] ];
```

En las consultas que utilizan GROUP BY se obtiene una fila por cada uno de los grupos producidos. Para ejecutar la cláusula GROUP BY se parte de las filas de la tabla que cumplen el predicado establecido en la cláusula WHERE y se agrupan en función de los valores comunes en la columna o columnas especificadas. Mediante la cláusula HAVING se realiza una restricción sobre los grupos obtenidos por la cláusula GROUP BY, y se seleccionan aquellos que cumplen el predicado establecido en la condición.

Evidentemente, en la condición de la cláusula HAVING sólo pueden aparecer restricciones sobre columnas por las que se ha agrupado y también funciones de grupo sobre cualquier otra columna de la tabla. Lo mismo sucede en la cláusula SELECT: sólo es posible especificar de manera directa columnas que aparecen en la cláusula GROUP BY y también funciones de grupo sobre cualquier otra columna. Cuando en las cláusulas SELECT o HAVING aparecen columnas que no se han especificado en la cláusula GROUP BY y que tampoco están afectadas por una función de grupo, se produce un error.

4.6.4. Ejemplos

Ejemplo 4.4 *Se quiere obtener el importe medio por factura, sin tener en cuenta los descuentos ni el IVA.*

El importe medio por factura se calcula obteniendo primero la suma del importe de todas las facturas y dividiendo después el resultado entre el número total de facturas. La suma del importe de todas las facturas se obtiene sumando el importe de todas las líneas de factura. El importe de cada línea se calcula multiplicando el número de unidades pedidas (cant) por el precio unitario (precio).

Por lo tanto, la solución a este ejercicio es la siguiente:

```
SELECT ROUND(SUM(cant*precio)/COUNT(DISTINCT codfac),2)
       AS importe_medio
FROM   lineas_fac;
```

Se ha redondeado a dos decimales porque el resultado es una cantidad en euros.

Ejemplo 4.5 *Se quiere obtener la fecha de la primera factura del cliente cuyo código es el 210, la fecha de su última factura (la más reciente) y el número de días que han pasado entre ambas facturas.*

Como se ha comentado antes, algunas funciones de columna se pueden utilizar también sobre las fechas. En general, las funciones `MIN` y `MAX` pueden usarse sobre todo aquel tipo de datos en el que haya definida una ordenación: tipos numéricos, cadenas y fechas.

Ambas funciones sirven, por lo tanto, para obtener la fecha de la primera y de la última factura. Restando ambas fechas se obtiene el número de días que hay entre ambas.

```
SELECT MIN(fecha) AS primera, MAX(fecha) AS ultima,
       MAX(fecha) - MIN(fecha) AS dias
FROM   facturas
WHERE  codcli = 210;
```

Ejemplo 4.6 *Se quiere obtener un listado con los clientes que tienen más de cinco facturas con 18 % de IVA, indicando cuántas de ellas tiene cada uno.*

Para resolver este ejercicio se deben tomar las facturas (tabla `FACTURAS`) y seleccionar aquellas con 18 % de IVA (`WHERE`). A continuación, se debe agrupar las facturas (`GROUP BY`) de manera que haya un grupo para cada cliente (columna `codcli`). Una vez formados los grupos, se deben seleccionar aquellos que contengan más de cinco facturas (`HAVING`). Por último, se debe mostrar (`SELECT`) el código de cada cliente y su número de facturas.

```
SELECT codcli, COUNT(*) AS facturas
FROM   facturas
WHERE  iva = 18
GROUP BY codcli
HAVING COUNT(*) > 5;
```

Ejemplo 4.7 *Se quiere obtener un listado con el número de facturas que hay en cada año, de modo que aparezca primero el año con más facturas. Además, para cada año se debe mostrar el número de clientes que han hecho compras y en cuántos días del año se han realizado éstas.*

```
SELECT EXTRACT(year FROM fecha) AS año,
       COUNT(*) AS nfacturas,
       COUNT(DISTINCT codcli) AS nclientes,
       COUNT(DISTINCT codven) AS nvendedores,
       COUNT(DISTINCT fecha) AS ndias
FROM   facturas
```

```
GROUP BY EXTRACT(year FROM fecha)
ORDER BY nfacturas DESC;
-- nfacturas es el nombre que se ha dado a COUNT(*)
```

Como se ve en el ejemplo, es posible utilizar expresiones en la cláusula `GROUP BY`. El ejemplo también muestra cómo se puede hacer referencia a los nombres con que se renombran las expresiones del `SELECT`, en la cláusula `ORDER BY`. Esto es así porque la cláusula `ORDER BY` es la única que se ejecuta tras el `SELECT`.

Ejemplo 4.8 *De los clientes cuyo código está entre el 240 y el 250, mostrar el número de facturas que cada uno tiene con cada IVA distinto.*

```
SELECT codcli, COALESCE(iva,0) AS iva, COUNT(*) AS facturas
FROM   facturas
WHERE  codcli BETWEEN 240 AND 250
GROUP BY codcli, COALESCE(iva,0);
```

Para resolver el ejercicio, se han agrupado las facturas teniendo en cuenta dos criterios: el cliente y el IVA. De este modo, quedan en el mismo grupo las facturas que son de un mismo cliente y con un mismo tipo de IVA. Puesto que en la base de datos con que se trabaja se debe interpretar el IVA nulo como cero, se ha utilizado la función `COALESCE`. Si no se hubiera hecho esto, las facturas de cada cliente con IVA nulo habrían dado lugar a un nuevo grupo (distinto del de IVA cero), ya que la cláusula `GROUP BY` no ignora los nulos sino que los toma como si fueran todos un mismo valor.

4.6.5. Algunas cuestiones importantes

A continuación se plantean algunas cuestiones que es importante tener en cuenta cuando se realizan agrupaciones:

- Cuando se utilizan funciones de grupo en la cláusula `SELECT` sin que haya `GROUP BY`, el resultado de ejecutar la consulta tiene una sola fila.
- A diferencia del resto de funciones que proporciona SQL, las funciones de grupo sólo se utilizan en las cláusulas `SELECT` y `HAVING`, nunca en la cláusula `WHERE`.
- La sentencia `SELECT` tiene dos cláusulas para realizar restricciones: `WHERE` y `HAVING`. Es muy importante saber situar cada restricción en su lugar: las restricciones que se deben realizar a nivel de filas, se sitúan en la cláusula `WHERE`; las restricciones que se deben realizar sobre grupos (normalmente involucran funciones de grupo), se sitúan en la cláusula `HAVING`.

- El modificador DISTINCT puede ser necesario en la cláusula SELECT de una sentencia que tiene GROUP BY sólo cuando las columnas que se muestren en la cláusula SELECT no sean todas las que aparecen en la cláusula GROUP BY.

- Una vez formados los grupos mediante la cláusula GROUP BY (son grupos de filas, no hay que olvidarlo), del contenido de cada grupo sólo es posible conocer el valor de las columnas por las que se ha agrupado (ya que dentro del grupo, todas las filas tienen dichos valores en común), por lo que sólo estas columnas son las que pueden aparecer, directamente, en las cláusulas SELECT y HAVING. Además, en estas cláusulas, se pueden incluir funciones de grupo que actúen sobre las columnas que no aparecen en la cláusula GROUP BY.

4.7. Subconsultas

Una subconsulta es una sentencia SELECT anidada en otra sentencia SQL, que puede ser otra SELECT o bien cualquier sentencia de manejo de datos (INSERT, UPDATE, DELETE). Las subconsultas pueden anidarse unas dentro de otras tanto como sea necesario (cada SGBD puede tener un nivel máximo de anidamiento, que difícilmente se alcanzará). En este apartado se muestra cómo el uso de subconsultas en las cláusulas WHERE y HAVING otorga mayor potencia para la realización de restricciones. Además, en este apartado se introduce el uso de subconsultas en la cláusula FROM.

4.7.1. Subconsultas en la cláusula WHERE

La cláusula WHERE se utiliza para realizar restricciones a nivel de filas. El predicado que se evalúa para realizar una restricción está formado por comparaciones unidas por los operadores AND/OR. Cada comparación involucra dos operandos que pueden ser:

(a) Dos columnas de la tabla sobre la que se realiza la consulta.

```
-- artículos cuyo stock es el mínimo deseado
SELECT *
FROM   articulos
WHERE  stock = stock_min;
```

(b) Una columna de la tabla de la consulta y una constante.

```
-- artículos cuya descripción empieza como se indica
SELECT *
FROM   articulos
WHERE  UPPER(descrip) LIKE 'PROLONG%';
```

(c) Una columna o una constante y una subconsulta sobre alguna tabla de la base de datos.

```
-- artículos vendidos con descuento mayor del 45%
SELECT *
FROM   articulos
WHERE  codart IN ( SELECT codart FROM lineas_fac
                   WHERE dto > 45 );
```

Además de los dos operandos, cada comparación se realiza con un operador. Hay una serie de operadores que se pueden utilizar con las subconsultas para establecer predicados en las restricciones. Son los que se muestran a continuación:

`expresión operador ( subconsulta )`

En este predicado la subconsulta debe devolver un solo valor (una fila con una columna). El predicado se evalúa a verdadero si la comparación indicada por el `operador` (`=, <>, >, <, >=, <=`), entre el resultado de la `expresión` y el de la subconsulta, es verdadero. Si la subconsulta devuelve más de un valor (una columna con varias filas o más de una columna), se produce un error de ejecución.

```
-- facturas con descuento máximo
SELECT *
FROM   facturas
WHERE  dto = ( SELECT MAX(dto) FROM facturas );
```

`(expr1, expr2, ...) operador ( subconsulta )`

En un predicado de este tipo, la subconsulta debe devolver una sola fila y tantas columnas como las especificadas entre paréntesis a la izquierda del `operador` (`=, <>, >, <, >=, <=`).

Las expresiones de la izquierda `expr1, expr2, ...` se evalúan y la fila que forman se compara, utilizando el `operador`, con la fila que devuelve la subconsulta.

El predicado se evalúa a verdadero si el resultado de la comparación es verdadero para la fila devuelta por la subconsulta. En caso contrario, se evalúa a falso. Si la subconsulta no devuelve ninguna fila, se evalúa a nulo.[2]

Dos filas se consideran iguales si los atributos correspondientes son iguales y no nulos en ambas; se consideran distintas si algún atributo es distinto en ambas filas y no nulo. En cualquier otro caso, el resultado del predicado es nulo.

[2]Hay que tener en cuenta que una restricción se cumple si el resultado de su predicado es verdadero; si el predicado es falso o nulo, se considera que la restricción no se cumple.

Si la subconsulta devuelve más de una fila, se produce un error de ejecución.

```
-- facturas con descuento máximo e IVA máximo
SELECT *
FROM   facturas
WHERE  (dto, iva) =
       ( SELECT MAX(dto), MAX(iva) FROM facturas );
```

`expresión IN ( subconsulta )`
El operador `IN` ya ha sido utilizado anteriormente, especificando una lista de valores entre paréntesis. Otro modo de especificar esta lista de valores es incluyendo una subconsulta que devuelva una sola columna. En este caso, el predicado se evalúa a verdadero si el resultado de la `expresión` es igual a alguno de los valores de la columna devuelta por la subconsulta.

El predicado se evalúa a falso si no se encuentra ningún valor en la subconsulta que sea igual a la `expresión`; cuando la subconsulta no devuelve ninguna fila, también se evalúa a falso.

Si el resultado de la `expresión` es un nulo, o ninguno de los valores de la subconsulta es igual a la `expresión` y la subconsulta ha devuelto algún nulo, el predicado se evalúa a nulo.

```
-- pueblos en donde hay algún cliente
SELECT codpue, nombre
FROM   pueblos
WHERE  codpue IN ( SELECT codpue FROM clientes);
```

`(expr1, expr2, ...) IN ( subconsulta )`
En este predicado la subconsulta debe devolver tantas columnas como las especificadas entre paréntesis a la izquierda del operador `IN`.

Las expresiones de la izquierda `expr1, expr2, ...` se evalúan y la fila que forman se compara con las filas de la subconsulta, una a una.

El predicado se evalúa a verdadero si se encuentra alguna fila igual en la subconsulta. En caso contrario se evalúa a falso (incluso si la subconsulta no devuelve ninguna fila).

Dos filas se consideran iguales si los atributos correspondientes son iguales y no nulos en ambas; se consideran distintas si algún atributo es distinto en ambas filas y no nulo. En cualquier otro caso, el resultado del predicado es nulo.

Si la subconsulta devuelve alguna fila de nulos y el resto de las filas son distintas de la fila de la izquierda del operador `IN`, el predicado se evalúa a nulo.

```
-- clientes que han comprado en algún mes en
-- que ha comprado el cliente con código 282
SELECT DISTINCT codcli
FROM   facturas
WHERE  ( EXTRACT(month FROM fecha),
         EXTRACT(year FROM fecha) )
       IN ( SELECT EXTRACT(month FROM fecha),
                   EXTRACT(year FROM fecha)
            FROM   facturas
            WHERE  codcli = 282);
```

`expresión NOT IN ( subconsulta )`

Cuando `IN` va negado, el predicado se evalúa a verdadero si la `expresión` es distinta de todos los valores de la columna devuelta por la subconsulta.

También se evalúa a verdadero cuando la subconsulta no devuelve ninguna fila. Si se encuentra algún valor igual a la `expresión`, se evalúa a falso.

Si el resultado de la `expresión` es un nulo, o si la subconsulta devuelve algún nulo y valores distintos a la `expresión`, el predicado se evalúa a nulo.

```
-- número de clientes que no tienen facturas
SELECT COUNT(*)
FROM   clientes
WHERE  codcli NOT IN ( SELECT codcli
                       FROM   facturas
                       WHERE  codcli IS NOT NULL );
```

Nótese que en el ejemplo se ha incluido la restricción `codcli IS NOT NULL` en la subconsulta porque la columna `FACTURAS.codcli` acepta nulos. Un nulo en esta columna haría que el predicado `NOT IN` se evaluara a nulo para todos los clientes de la consulta principal.

`(expr1, expr2, ...) NOT IN ( subconsulta )`

En este predicado, la subconsulta debe devolver tantas columnas como las especificadas entre paréntesis a la izquierda del operador `NOT IN`. Las expresiones de la izquierda `expr1, expr2, ...` se evalúan y la fila que forman se compara con las filas de la subconsulta, fila a fila.

El predicado se evalúa a verdadero si no se encuentra ninguna fila igual en la subconsulta. También se evalúa a verdadero si la subconsulta no devuelve ninguna fila. Si se encuentra alguna fila igual, se evalúa a falso.

Dos filas se consideran iguales si los atributos correspondientes son iguales y no nulos en ambas; se consideran distintas si algún atributo es

distinto en ambas filas y no nulo. En cualquier otro caso, el resultado del predicado es nulo.

Si la subconsulta devuelve alguna fila de nulos y el resto de las filas son distintas de la fila de la izquierda del operador `NOT IN`, el predicado se evalúa a nulo.

```
-- clientes que no tienen facturas con IVA y dto
-- como tienen los clientes del rango especificado
SELECT DISTINCT codcli
FROM   facturas
WHERE  ( COALESCE(iva,0), COALESCE(dto,0) )
       NOT IN ( SELECT COALESCE(iva,0), COALESCE(dto,0)
                FROM   facturas
                WHERE  codcli BETWEEN 171 AND 174);
```

`expresión operador ANY ( subconsulta )`

En este uso de `ANY` la subconsulta debe devolver una sola columna. El `operador` es una comparación (`=, <>, >, <, >=, <=`).

El predicado se evalúa a verdadero si la comparación establecida por el `operador` es verdadera para alguno de los valores de la columna devuelta por la subconsulta. En caso contrario se evalúa a falso.

```
-- facturas con IVA como los de las facturas sin dto
SELECT *
FROM   facturas
WHERE  iva = ANY( SELECT iva
                  FROM facturas
                  WHERE COALESCE(dto,0) = 0 );
```

Si la subconsulta no devuelve ninguna fila, devuelve falso. Si ninguno de los valores de la subconsulta coincide con la expresión de la izquierda del operador y en la subconsulta se ha devuelto algún nulo, se evalúa a nulo.

En lugar de `ANY` puede aparecer `SOME`, son sinónimos. El operador `IN` es equivalente a `= ANY`.

`(expr1, expr2, ...) operador ANY ( subconsulta )`

En este uso de `ANY` la subconsulta debe devolver tantas columnas como las especificadas entre paréntesis a la izquierda del operador. Las expresiones de la izquierda `expr1, expr2, ...` se evalúan y la fila que forman se compara con las filas de la subconsulta, fila a fila.

El predicado se evalúa a verdadero si la comparación establecida por el operador es verdadera para alguna de las filas devueltas por la subconsulta. En caso contrario se evalúa a falso (incluso si la subconsulta no devuelve ninguna fila).

Dos filas se consideran iguales si los atributos correspondientes son iguales y no nulos en ambas; se consideran distintas si algún atributo es distinto en ambas filas y no nulo. En cualquier otro caso, el resultado del predicado es nulo.

Si la subconsulta devuelve alguna fila de nulos, el predicado no podrá ser falso (será verdadero o nulo).

```
-- clientes que han comprado algún mes
-- en que ha comprado el cliente especificado
SELECT DISTINCT codcli
FROM   facturas
WHERE  ( EXTRACT(month FROM fecha),
         EXTRACT(year FROM fecha) )
       = ANY( SELECT EXTRACT(month FROM fecha),
                     EXTRACT(year FROM fecha)
              FROM   facturas
              WHERE  codcli = 282);
```

En lugar de `ANY` puede aparecer `SOME`.

`expresión operador ALL ( subconsulta )`

En este uso de `ALL` la subconsulta debe devolver una sola columna. El `operador` es una comparación (`=, <>, >, <, >=, <=`).

El predicado se evalúa a verdadero si la comparación establecida por el operador es verdadera para todos los valores de la columna devuelta por la subconsulta. También se evalúa a verdadero cuando la subconsulta no devuelve ninguna fila. En caso contrario se evalúa a falso. Si la subconsulta devuelve algún nulo, el predicado se evalúa a nulo

```
-- facturas con descuento máximo
SELECT *
FROM   facturas
WHERE  dto >= ALL ( SELECT COALESCE(dto,0)
                    FROM facturas );
```

Nótese que, si en el ejemplo anterior, la subconsulta no utiliza `COALESCE` para convertir los descuentos nulos en descuentos cero, la consulta principal no devuelve ninguna fila porque al haber nulos en el resultado de la subconsulta, el predicado se evalúa a nulo.

El operador `NOT IN` es equivalente a `<>ALL`.

`(expr1, expr2, ...) operador ALL ( subconsulta )`

En este uso de `ALL`, la subconsulta debe devolver tantas columnas como las especificadas entre paréntesis a la izquierda del `operador`.

Las expresiones de la izquierda `expr1, expr2, ...` se evalúan y la fila que forman se compara con las filas de la subconsulta, fila a fila.

El predicado se evalúa a verdadero si la comparación establecida por el operador es verdadera para todas las filas devueltas por la subconsulta; cuando la subconsulta no devuelve ninguna fila también se evalúa a verdadero. En caso contrario se evalúa a falso.

Dos filas se consideran iguales si los atributos correspondientes son iguales y no nulos en ambas; se consideran distintas si algún atributo es distinto en ambas filas y no nulo. En cualquier otro caso, el resultado del predicado es nulo.

Si la subconsulta devuelve alguna fila de nulos, el predicado no podrá ser verdadero (será falso o nulo).

```
-- muestra los datos del cliente especificado si
-- siempre ha comprado sin descuento y con 18% de IVA
SELECT *
FROM   clientes
WHERE  codcli = 162
AND    ( 18, 0 ) =
       ALL (SELECT COALESCE(iva,0), COALESCE(dto,0)
            FROM   facturas
            WHERE  codcli = 162 );
```

Cuando se utilizan subconsultas en predicados, el SGBD no obtiene el resultado completo de la subconsulta, a menos que sea necesario. Lo que hace es ir obteniendo filas de la subconsulta hasta que es capaz de determinar si el predicado es verdadero.

4.7.2. Subconsultas en la cláusula HAVING

La cláusula `HAVING` permite hacer restricciones sobre grupos y necesariamente va precedida de una cláusula `GROUP BY`. Para hacer este tipo de restricciones también es posible incluir subconsultas cuando sea necesario.

La siguiente consulta obtiene el código del pueblo que tiene más clientes:

```
SELECT codpue
FROM   clientes
GROUP BY codpue
HAVING COUNT(*) >= ALL ( SELECT COUNT(*)
                         FROM   clientes
                         GROUP BY codpue );
```

En primer lugar se ejecuta la subconsulta, obteniéndose una columna de números en donde cada uno indica el número de clientes en cada pueblo. La

subconsulta se sustituye entonces por los valores de esta columna, por ejemplo:

```
SELECT codpue
FROM   clientes
GROUP BY codpue
HAVING COUNT(*) >= ALL (1,4,7,9,10);
```

Por último, se ejecuta la consulta principal. Para cada grupo se cuenta el número de clientes que tiene. Pasan la restricción del `HAVING` aquel o aquellos pueblos que en esa cuenta tienen el máximo valor.

4.7.3. Subconsultas en la cláusula `FROM`

También es posible incluir subconsultas en la cláusula `FROM`, aunque en este caso no se utilizan para construir predicados sino para realizar una consulta sobre la tabla que se obtiene como resultado de ejecutar otra consulta. Siempre que se utilice una subconsulta en el `FROM` se debe dar un nombre a la tabla resultado mediante la cláusula `AS`.

```
SELECT COUNT(*), MAX(ivat), MAX(dtot)
FROM   ( SELECT DISTINCT COALESCE(iva,0) AS ivat,
                         COALESCE(dto,0) AS dtot
         FROM   facturas ) AS t;
```

La consulta anterior cuenta las distintas combinaciones de IVA y descuento y muestra el valor máximo de éstos. Nótese que se han renombrado las columnas de la subconsulta para poder referenciarlas en la consulta principal. Esta consulta no se puede resolver si no es de este modo ya que `COUNT` no acepta una lista de columnas como argumento.

4.7.4. Ejemplos

Ejemplo 4.9 *Se quiere obtener los datos completos del cliente al que pertenece la factura 5886.*

Para dar la respuesta podemos hacerlo en dos pasos, es decir, con dos consultas separadas:

```
SELECT codcli FROM facturas WHERE codfac = 5886;

 codcli
--------
    264

SELECT *
FROM   clientes
WHERE  codcli = 264;
```

Puesto que es posible anidar las sentencias SELECT para obtener el resultado con una sola consulta, una solución que obtiene el resultado en un solo paso es la siguiente:

```
SELECT *
FROM   clientes
WHERE  codcli = ( SELECT codcli
                  FROM facturas WHERE codfac = 5886 );
```

Se ha utilizado el operador de comparación = porque se sabe con certeza que la subconsulta devuelve un solo código de cliente, ya que la condición de búsqueda es de igualdad sobre la clave primaria de la tabla del FROM.

Ejemplo 4.10 *Se quiere obtener los datos completos de los clientes que tienen facturas en agosto del año pasado. El resultado se debe mostrar ordenado por el nombre del cliente.*

De nuevo se puede dar la respuesta en dos pasos:

```
SELECT codcli FROM facturas
WHERE EXTRACT(month FROM fecha)=8
AND   EXTRACT(year FROM fecha) =
      EXTRACT(year FROM CURRENT_DATE)-1;

 codcli
--------
    105
     12
      .
      .
      .
    342
    309
    357

SELECT *
FROM   clientes
WHERE  codcli IN (105,12,...,342,309,357);
```

Se ha utilizado el operador IN porque la primera consulta devuelve varias filas. Esto debe saberse sin necesidad de probar la sentencia. Como esta vez no se seleccionan las facturas por una columna única (clave primaria o clave alternativa), es posible que se obtengan varias filas y por lo tanto se debe utilizar IN.

Tal y como se ha hecho en el ejemplo anterior, ambas sentencias pueden integrarse en una sola:

```
SELECT *
FROM   clientes
WHERE  codcli IN ( SELECT codcli FROM facturas
                   WHERE EXTRACT(month FROM fecha)=8
                   AND   EXTRACT(year FROM fecha) =
                         EXTRACT(year FROM CURRENT_DATE)-1 )
ORDER BY nombre;
```

4.7.5. Algunas cuestiones importantes

A continuación se plantean algunas cuestiones que es importante tener en cuenta cuando se realizan subconsultas:

- Las subconsultas utilizadas en predicados del tipo `expresión operador (subconsulta)` o `(expr1, expr2, ...) operador (subconsulta)` deben devolver siempre una sola fila; en otro caso, se producirá un error. Si la subconsulta ha de devolver varias filas se debe utilizar `IN`, `NOT IN`, `operador ANY`, `operador ALL`.

- Es importante ser cuidadosos con las subconsultas que pueden devolver nulos. Una restricción se supera si el predicado se evalúa a verdadero; no se supera si se evalúa a falso o a nulo. Dos casos que no conviene olvidar son los siguientes:

  - `NOT IN` se evalúa a verdadero cuando la subconsulta no devuelve ninguna fila; si la subconsulta devuelve un nulo/fila de nulos, se evalúa a nulo.

  - `operador ALL` se evalúa a verdadero cuando la subconsulta no devuelve ninguna fila; si la subconsulta devuelve un nulo/fila de nulos, se evalúa a nulo.

- Cuando se utilizan subconsultas en la cláusula `FROM` es preciso renombrar las columnas del `SELECT` de la subconsulta que son expresiones. De ese modo, será posible hacerles referencia en la consulta principal. Además, la tabla resultado de la subconsulta también se debe renombrar en el `FROM` de la consulta principal.

4.8. Consultas multitabla

En este apartado se muestra cómo hacer consultas que involucran a datos de varias tablas. Aunque mediante las subconsultas se ha conseguido realizar consultas de este tipo, aquí se verá que en ocasiones, es posible escribir consultas equivalentes que no hacen uso de subconsultas y que se ejecutan de modo más eficiente. El operador que se introduce es la *concatenación* (`JOIN`).

4.8.1. La concatenación: `JOIN`

La concatenación es una de las operaciones más útiles del lenguaje SQL. Esta operación permite combinar información de varias tablas sin necesidad de utilizar subconsultas para ello.

La concatenación natural (`NATURAL JOIN`) de dos tablas `R` y `S` obtiene como resultado una tabla cuyas filas son todas las filas de `R` concatenadas con todas las filas de `S` que en las columnas que se llaman igual tienen los mismos valores. Las columnas por las que se hace la concatenación aparecen una sola vez en el resultado.

La siguiente sentencia hace una concatenación natural de las tablas `FACTURAS` y `CLIENTES`. Ambas tablas tienen una columna con el mismo nombre, `codcli`, siendo `FACTURAS.codcli` una clave ajena a `CLIENTES.codcli` (clave primaria).

```
SELECT *
FROM   facturas NATURAL JOIN clientes;
```

Según la definición de la operación `NATURAL JOIN`, el resultado tendrá las siguientes columnas: `codfac, fecha, codven, iva, dto, codcli, nombre, direccion, codpostal, codpue`. En el resultado de la concatenación cada fila representa una factura que cuenta con sus datos (la cabecera) y los datos del cliente al que pertenece. Si alguna factura tiene `codcli` nulo, no aparece en el resultado de la concatenación puesto que no hay ningún cliente con el que pueda concatenarse.

Cambiando el contenido de la cláusula `SELECT`, cambia el resultado de la consulta. Por ejemplo:

```
SELECT DISTINCT codcli, nombre, direccion, codpostal, codpue
FROM   facturas NATURAL JOIN clientes;
```

Esta sentencia muestra los datos de los clientes que tienen facturas. Puesto que se ha hecho la concatenación, si hay clientes que no tienen facturas, no se obtienen en el resultado ya que no tienen ninguna factura con la que concatenarse.

A continuación, se desea modificar la sentencia anterior para que se obtenga también el nombre de la población del cliente. Se puede pensar que el nombre de la población se puede mostrar tras hacer una concatenación natural con la

tabla `PUEBLOS`. El objetivo es concatenar cada cliente con su población a través de la clave ajena `codpue`. Sin embargo, la concatenación natural no es útil en este caso porque las tablas `PUEBLOS` y `CLIENTES` tienen también otra columna que se llama igual: la columna `nombre`. `CLIENTES.nombre` contiene el nombre de cada cliente y `PUEBLOS.nombre` contiene el nombre de cada pueblo. Ambos nombres no significan lo mismo, por lo que la concatenación natural a través de ellas no permite obtener el resultado que se desea.

¿Qué se obtendrá como resultado al ejecutar la siguiente sentencia?

```
SELECT *
FROM   facturas NATURAL JOIN clientes NATURAL JOIN pueblos;
```

Se obtendrán las facturas de los clientes cuyo nombre completo coincide con el nombre de su pueblo, cosa poco probable que suceda.

Cuando se quiere concatenar varias tablas que tienen varios nombres de columnas en común y no todos han de utilizarse para realizar la concatenación, se puede disponer de la operación `INNER JOIN`, que permite especificar las columnas sobre las que hacer la operación mediante la cláusula `USING`.

```
SELECT DISTINCT codcli, clientes.nombre, codpue,
                pueblos.nombre
FROM   facturas INNER JOIN clientes USING (codcli)
                INNER JOIN pueblos  USING (codpue);
```

Nótese que, en la consulta anterior, algunas columnas van precedidas por el nombre de la tabla a la que pertenecen. Esto es necesario cuando hay columnas que se llaman igual en el resultado: se especifica el nombre de la tabla para evitar ambigüedades. Esto sucede cuando las tablas que se concatenan tienen nombres de columnas en común y la concatenación no se hace a través de ellas, como ha sucedido en el ejemplo con las columnas `CLIENTES.nombre` y `PUEBLOS.nombre`. En el resultado hay dos columnas `nombre` y, sin embargo, una sola columna `codcli` y una sola columna `codpue` (estas dos últimas aparecen sólo una vez porque las concatenaciones se han hecho a través de ellas).

En realidad, en SQL el nombre de cada columna está formado por el nombre de su tabla, un punto y el nombre de la columna (`FACTURAS.iva`, `CLIENTES.nombre`). Por comodidad, cuando no hay ambigüedad al referirse a una columna, se permite omitir el nombre de la tabla a la que pertenece, que es lo que se había estado haciendo hasta ahora en este capítulo.

Cuando las columnas por las que se hace la concatenación no se llaman igual en las dos tablas, se utiliza `ON` para especificar la condición de concatenación de ambas columnas, tal y como se ve en el siguiente ejemplo. En él se introduce también el uso de alias para las tablas, lo que permite no tener que escribir el nombre completo para referirse a sus columnas:

```
SELECT v.codven, v.nombre AS vendedor,
       j.codven AS codjefe, j.nombre AS jefe
```

```
FROM   vendedores AS v INNER JOIN vendedores AS j
       ON (v.codjefe=j.codven);
```

Esta sentencia obtiene el código y el nombre de cada vendedor, junto al código y el nombre del vendedor que es su jefe.

Es aconsejable utilizar siempre alias para las tablas cuando se hagan consultas multitabla, y utilizarlos para especificar todas las columnas, aunque no haya ambigüedad. Es una cuestión de estilo.

Ya que este tipo de concatenación (`INNER JOIN`) es el más habitual, se permite omitir la palabra `INNER` al especificarlo, tal y como se muestra en el siguiente ejemplo:

```
SELECT DISTINCT c.codcli, c.nombre, c.codpue, p.nombre
FROM   facturas AS f JOIN clientes AS c USING (codcli)
                     JOIN pueblos  AS p USING (codpue)
WHERE  COALESCE(f.iva,0) = 18
AND    COALESCE(f.dto,0) = 0;
```

Aunque la operación de `NATURAL JOIN` es la que originalmente se definió en el modelo relacional, su uso en SQL no es aconsejable puesto que la creación de nuevas columnas en tablas de la base de datos puede dar lugar a errores en las sentencias que las consultan, si estas nuevas columnas tienen el mismo nombre que otras columnas de otras tablas con las que se han de concatenar.

Es recomendable, al construir las concatenaciones, especificar las tablas en el mismo orden en el que aparecen en el diagrama referencial (figura 4.2). De este modo será más fácil depurar las sentencias, así como identificar qué hace cada una: en el resultado de una consulta escrita de este modo, cada fila representará lo mismo que representa cada fila de la primera tabla que aparezca en la cláusula `FROM` y en este resultado habrá, como mucho, tantas filas como filas hay en dicha tabla.

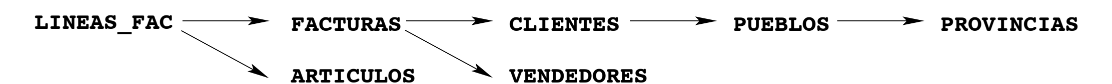


Figura 4.2: Diagrama referencial de la base de datos.

Hay un aspecto que todavía no se ha tenido en cuenta: los nulos en las columnas a través de las cuales se realizan las concatenaciones. Por ejemplo, si se quiere obtener un listado con las facturas del mes de diciembre del año pasado, donde aparezcan los nombres del cliente y del vendedor, se puede escribir la siguiente consulta:

```
SELECT f.codfac, f.fecha, f.codcli, c.nombre,
       f.codven, v.nombre
FROM   facturas AS f JOIN clientes AS c USING (codcli)
                     JOIN vendedores AS v USING (codven)
WHERE  EXTRACT(month FROM f.fecha) = 12
AND    EXTRACT(year FROM f.fecha) =
       EXTRACT(year FROM CURRENT_DATE)-1;
```

De todas las facturas que hay en dicho mes, aparecen en el resultado sólo algunas. Esto es debido a que las columnas `FACTURAS.codcli` y `FACTURAS.codven` aceptan nulos. Las facturas con algún nulo en alguna de estas columnas son las que no aparecen en el resultado.

Para evitar estos problemas, se puede hacer uso de la operación `OUTER JOIN` con tres variantes: `LEFT, RIGHT, FULL`. Con `LEFT/RIGHT OUTER JOIN`, en el resultado se muestran todas las filas de la tabla de la izquierda/derecha; aquellas que no tienen nulos en la columna de concatenación, se concatenan con las filas de la otra tabla mediante `INNER JOIN`. Las filas de la tabla de la izquierda/derecha que tienen nulos en la columna de concatenación aparecen en el resultado concatenadas con una fila de nulos. Con `FULL OUTER JOIN` se hacen ambas operaciones: `LEFT OUTER JOIN` y `RIGHT OUTER JOIN`.

Teniendo en cuenta que, tanto `FACTURAS.codcli` como `FACTURAS.codven` aceptan nulos, el modo correcto de realizar la consulta en este último ejemplo será:

```
SELECT f.codfac, f.fecha, f.codcli, c.nombre,
       f.codven, v.nombre
FROM   facturas AS f
           LEFT OUTER JOIN clientes AS c USING (codcli)
           LEFT OUTER JOIN vendedores AS v USING (codven)
WHERE  EXTRACT(month FROM f.fecha) = 12
AND    EXTRACT(year FROM f.fecha) =
       EXTRACT(year FROM CURRENT_DATE)-1;
```

Como se ha visto, el `OUTER JOIN` tiene sentido cuando no se quiere perder filas en una concatenación en que una de las columnas que interviene acepta nulos. Otro caso en que esta operación tiene sentido es cuando las filas de una tabla no tienen filas para concatenarse en la otra tabla porque no son referenciadas por ninguna de ellas. Es el caso del siguiente ejemplo:

```
SELECT c.codcli, c.nombre, COUNT(f.codfac) AS nfacturas
FROM   facturas AS f
           RIGHT OUTER JOIN clientes AS c USING (codcli)
GROUP BY c.codcli, c.nombre
ORDER BY 3 DESC;
```

Esta sentencia obtiene un listado con todos los clientes de la tabla CLIENTES y el número de facturas que cada uno tiene. Si algún cliente no tiene ninguna factura (no es referenciado por ninguna fila de la tabla de FACTURAS), también aparecerá en el resultado y la cuenta del número de facturas será cero. Nótese que la cuenta del número de facturas se hace sobre la clave primaria de FACTURAS (COUNT(f.codfac)) ya que los clientes sin facturas tienen un nulo en esta columna tras la concatenación y las funciones de columna ignoran los nulos, por lo que la cuenta será cero.

4.8.2. Sintaxis original de la concatenación

En versiones anteriores del estándar de SQL la concatenación no se realizaba mediante JOIN, ya que esta operación no estaba implementada directamente. En el lenguaje teórico en el que se basa SQL, el álgebra relacional, la operación de concatenación sí existe, pero ya que no es una operación primitiva, no fue implementada en SQL en un principio. No es una operación primitiva porque se puede llevar a cabo mediante la combinación de otras dos operaciones: el producto cartesiano y la restricción. La restricción se lleva a cabo mediante la cláusula WHERE, que ya es conocida. El producto cartesiano se lleva a cabo separando las tablas involucradas por una coma en la cláusula FROM, tal y como se muestra a continuación:

```
SELECT *
FROM   facturas, clientes;
```

La sentencia anterior combina todas las filas de la tabla facturas con todas las filas de la tabla clientes. Si la primera tiene n filas y la segunda tiene m filas, el resultado tendrá $n \times m$ filas.

Para hacer la concatenación de cada factura con el cliente que la ha solicitado, se debe hacer una restricción: de las $n \times m$ filas hay que seleccionar aquellas en las que coinciden los valores de las columnas codcli.

```
SELECT *
FROM   facturas, clientes
WHERE  facturas.codcli = clientes.codcli;
```

La siguiente consulta, que utiliza el formato original para realizar las concatenaciones. Obtiene los datos de las facturas con 18 % de IVA y sin descuento, con el nombre del cliente:

```
SELECT facturas.codfac, facturas.fecha,
       facturas.codcli, clientes.nombre, facturas.codven
FROM   facturas, clientes
WHERE  facturas.codcli = clientes.codcli -- concatenación
AND    COALESCE(facturas.iva,0) = 18     -- restricción
AND    COALESCE(facturas.dto,0) = 0;     -- restricción
```

No hay que olvidar que la concatenación que se acaba de mostrar utiliza una sintaxis que ha quedado obsoleta en el estándar de SQL. La sintaxis del estándar actual es más aconsejable porque permite identificar más claramente qué son restricciones (aparecerán en el `WHERE`) y qué son condiciones de concatenación (aparecerán en el `FROM` con la palabra clave `JOIN`). Sin embargo, es importante conocer esta sintaxis porque todavía es muy habitual su uso.

4.8.3. Ejemplos

Ejemplo 4.11 *Obtener los datos completos del cliente al que pertenece la factura 5886.*

Una versión que utiliza subconsultas es la siguiente:

```
SELECT *
FROM   clientes
WHERE  codcli = ( SELECT codcli
                  FROM facturas WHERE codfac = 5886 );
```

Una versión que utiliza `JOIN` es la siguiente:

```
SELECT c.*
FROM   facturas f JOIN clientes c USING (codcli)
WHERE  f.codfac = 5886;
```

Ejemplo 4.12 *Obtener el código de las facturas en las que se ha pedido el artículo que tiene actualmente el precio más caro.*

Una versión en donde se utiliza el `JOIN` de subconsultas en el `FROM` es la siguiente:

```
SELECT DISTINCT l.codfac
FROM   lineas_fac AS l JOIN articulos AS a USING (codart)
       JOIN (SELECT MAX(precio) AS precio
             FROM articulos) AS t ON (a.precio = t.precio);
```

En la siguiente versión se utiliza la subconsulta para hacer una restricción.

```
SELECT DISTINCT l.codfac
FROM   lineas_fac AS l JOIN articulos AS a USING (codart)
WHERE  a.precio =  (SELECT MAX(precio)
                    FROM articulos) ;
```

A continuación se muestra una versión que utiliza sólo subconsultas:

```
SELECT DISTINCT codfac
FROM   lineas_fac
WHERE  codart IN
           (SELECT codart
            FROM   articulos
            WHERE  precio =
                       (SELECT MAX(precio) FROM articulos));
```

Ejemplo 4.13 *Para cada vendedor de la provincia de Castellón, mostrar su nombre y el nombre de su jefe inmediato.*

```
SELECT v.codven, v.nombre AS vendedor, j.nombre AS jefe
FROM   vendedores AS v JOIN vendedores AS j
            ON (v.codjefe = j.codven)
       JOIN pueblos AS p ON (v.codpue = p.codpue)
WHERE  p.codpro = '12';
```

Nótese que ambas concatenaciones deben hacerse mediante `ON`: la primera porque las columnas de concatenación no tienen el mismo nombre, la segunda porque al concatenar con `PUEBLOS` hay dos columnas `codpue` en la tabla de la izquierda: `v.codpue` y `j.codven`.

4.8.4. Algunas cuestiones importantes

A continuación se plantean algunas cuestiones que es importante tener en cuenta cuando se realizan concatenaciones:

- Al hacer un `NATURAL JOIN` es importante fijarse muy bien en los nombres de las columnas de las tablas que participan en la operación. Como se sabe, mediante este operador se concatenan las filas de ambas tablas que en los atributos que tienen el mismo nombre tienen también los mismos valores. Por ejemplo, un `NATURAL JOIN` entre las tablas `PUEBLOS` y `CLIENTES` se realizará a través de las columnas `codpue` y `nombre`. El resultado, si contiene alguna fila, serán los datos de clientes que tienen como nombre el mismo nombre de su población. Si nuestro objetivo era realizar la concatenación a través de `codpue` podemos decir que el uso del `NATURAL JOIN` nos ha jugado una mala pasada. Concatenar filas por columnas no deseadas implica tener en cuenta más restricciones, con lo que los resultados obtenidos pueden no ser correctos. Es más aconsejable utilizar `INNER JOIN`, ya que pueden evitarse estos problemas al especificarse de manera explícita las columnas de concatenación.

- En la vida de una base de datos puede ocurrir que a una tabla se le deban añadir nuevas columnas para que pueda almacenar más información. Si esta tabla se ha utilizado para realizar algún `NATURAL JOIN` en alguna de las consultas de los programas de aplicación, hay que ser cuidadosos al escoger el nombre ya que si una nueva columna se llama igual que otra columna de la otra tabla participante en dicha operación, la concatenación que se hará ya no será la misma. Es posible evitar este tipo de problemas utilizando siempre `INNER JOIN` ya que éste requiere que se especifiquen las columnas por las que realizar la concatenación, y aunque se añadan nuevas columnas a las tablas, no cambiará la operación realizada por más que haya nuevas coincidencias de nombres en ambas tablas.
- Ordenar las tablas en el `FROM` tal y como aparecen en los diagramas referenciales ayuda a tener un mayor control de la consulta en todo momento: es posible saber si se ha olvidado incluir alguna tabla intermedia y es posible saber qué representa cada fila del resultado de la concatenación de todas las tablas implicadas. Además, será más fácil decidir qué incluir en la función `COUNT()` cuando sea necesaria, y también será más fácil determinar si en la proyección final (`SELECT`) es necesario el uso de `DISTINCT`.

4.9. Operadores de conjuntos

Los operadores de conjuntos del álgebra relacional son: el producto cartesiano, la unión, la intersección y la diferencia. El producto cartesiano se realiza en SQL especificando en la cláusula `FROM` las tablas involucradas en la operación, separadas por comas, tal y como se ha indicado anteriormente. A continuación se muestra cómo utilizar el resto de los operadores de conjuntos en las consultas en SQL.

La sintaxis para las uniones, intersecciones y diferencias es la siguiente:

```
sentencia_SELECT
UNION | INTERSECT | EXCEPT [ ALL ]
sentencia_SELECT
[ ORDER BY columna [ ASC | DESC ]
           [,columna [ ASC | DESC ] ];
```

Nótese que la cláusula `ORDER BY` sólo puede aparecer una vez en la consulta, al final de la misma. La ordenación se realizará sobre el resultado de la unión, intersección o diferencia.

Para poder utilizar cualquiera de estos tres nuevos operadores, las cabeceras de las sentencias `SELECT` involucradas deben devolver el mismo número de columnas, y las columnas correspondientes en ambas sentencias deberán ser del mismo tipo de datos.

4.9.1. Operador `UNION`

Este operador devuelve como resultado todas las filas que devuelve la primera sentencia `SELECT`, más aquellas filas de la segunda sentencia `SELECT` que no han sido ya devueltas por la primera. En el resultado no se muestran duplicados.

Se puede evitar la eliminación de duplicados especificando la palabra clave `ALL`. En este caso, si una fila aparece m veces en la primera sentencia y n veces en la segunda, en el resultado aparecerá $m + n$ veces.

Si se realizan varias uniones, éstas se evalúan de izquierda a derecha, a menos que se utilicen paréntesis para establecer un orden distinto.

La siguiente sentencia muestra los códigos de las poblaciones donde hay clientes o donde hay vendedores:

```
SELECT codpue FROM clientes
UNION
SELECT codpue FROM vendedores;
```

4.9.2. Operador `INTERSECT`

Este operador devuelve como resultado las filas que se encuentran tanto en el resultado de la primera sentencia `SELECT` como en el de la segunda sentencia `SELECT`. En el resultado no se muestran duplicados.

Se puede evitar la eliminación de duplicados especificando la palabra clave `ALL`. En este caso, si una misma fila aparece m veces en la primera sentencia y n veces en la segunda, en el resultado esta fila aparecerá $min(m, n)$ veces.

Si se realizan varias intersecciones, éstas se evalúan de izquierda a derecha, a menos que se utilicen paréntesis para establecer un orden distinto. La intersección tiene más prioridad, en el orden de evaluación, que la unión, es decir, `A UNION B INTERSECT C` se evalúa como `A UNION (B INTERSECT C)`.

La siguiente sentencia muestra los códigos de las poblaciones donde hay clientes y también hay vendedores:

```
SELECT codpue FROM clientes
INTERSECT
SELECT codpue FROM vendedores;
```

4.9.3. Operador `EXCEPT`

Este operador devuelve como resultado las filas que se encuentran en el resultado de la primera sentencia `SELECT` y no se encuentran en el resultado de la segunda sentencia `SELECT`. En el resultado no se muestran duplicados.

Se puede evitar la eliminación de duplicados especificando la palabra clave `ALL`. En este caso, si una misma fila aparece m veces en la primera sentencia y n veces en la segunda, en el resultado esta fila aparecerá $max(m-n, 0)$ veces.

Si se realizan varias diferencias, éstas se evalúan de izquierda a derecha, a menos que se utilicen paréntesis para establecer un orden distinto. La diferencia tiene la misma prioridad, en el orden de evaluación, que la unión.

La siguiente sentencia muestra los códigos de las poblaciones donde hay clientes y no hay vendedores:

```
SELECT codpue FROM clientes
EXCEPT
SELECT codpue FROM vendedores;
```

La diferencia no es una operación conmutativa, mientras que el resto de los operadores de conjuntos sí lo son.

4.9.4. Sentencias equivalentes

En muchas ocasiones, una misma consulta de datos puede responderse mediante distintas sentencias `SELECT` que utilizan operadores diferentes. Cada una de ellas dará, por lo general, un tiempo de respuesta diferente, y se puede considerar que una es mejor que otra en este aspecto.

El que una sentencia sea mejor en unas circunstancias no garantiza que vaya a serlo siempre: puede que al evolucionar el estado de la base de datos, una sentencia que era la mejor, deje de serlo porque las tablas hayan cambiado de tamaño o se haya creado o eliminado algún índice.

Es por todo lo anterior, que se considera importante que, ante una consulta de datos, sea posible obtener varias sentencias alternativas. En este apartado se presentan algunas equivalencias entre operadores que se pueden utilizar para obtener sentencias equivalentes.

- Una concatenación es equivalente a una expresión con el operador `IN` y una subconsulta. Dependiendo del número de filas que obtenga la subconsulta, será más o menos eficiente que la concatenación con `JOIN`.

- Una restricción con dos comparaciones unidas por `OR` es equivalente a la unión de dos sentencias `SELECT`, y sitúa cada una de estas comparaciones en una sentencia distinta.

- Una restricción con dos comparaciones unidas por `AND` es equivalente a la intersección de dos sentencias `SELECT`, y sitúa cada una de estas comparaciones en una sentencia distinta.

- Una restricción con dos comparaciones unidas por AND NOT es equivalente a la diferencia de dos sentencias SELECT, y sitúa la primera comparación en la primera sentencia y la segunda comparación en la segunda sentencia (conviene recordar que esta operación no es conmutativa).
- El operador NOT IN puede dar resultados inesperados cuando la subconsulta devuelve algún nulo. En general, es más aconsejable trabajar con operadores en positivo (sin NOT) (en el ejemplo que se ofrece después se verá el porqué). Una restricción con el operador NOT IN y una subconsulta, es equivalente a una restricción con IN y una subconsulta con EXCEPT.

4.9.5. Ejemplos

Ejemplo 4.14 *Obtener los datos de las poblaciones donde hay vendedores y no hay clientes.*

```
SELECT *
FROM   ( SELECT codpue FROM vendedores
         EXCEPT
         SELECT codpue FROM clientes ) AS t
       JOIN pueblos USING (codpue)
       JOIN provincias USING (codpro);
```

La tabla t contiene los códigos de las poblaciones en donde hay vendedores y no hay clientes. Tras concatenarla con PUEBLOS y PROVINCIAS se obtienen los datos completos de dichas poblaciones.

Ejemplo 4.15 *¿Cuántos clientes hay que entre todas sus facturas no tienen ninguna con 18 % de IVA?*

La siguiente solución utiliza el operador NOT IN. Nótese que es preciso tener en cuenta dos restricciones: la primera es que en la subconsulta del NOT IN se debe evitar los nulos, y la segunda es que hay que asegurarse de que los clientes seleccionados hayan realizado alguna compra (deben tener alguna factura).

```
SELECT COUNT(*) AS clientes
FROM   clientes
WHERE  codcli NOT IN ( SELECT codcli FROM facturas
                       WHERE  COALESCE(iva,0) = 18
                       AND    codcli IS NOT NULL )
AND    codcli IN (SELECT codcli FROM facturas);
```

Una sentencia equivalente sin NOT IN y que utiliza un operador de conjuntos, es la siguiente:

```
-- clientes con alguna factura
-- menos
-- clientes que tienen alguna con 18%
SELECT COUNT(*) AS clientes
FROM   ( SELECT codcli FROM facturas
         EXCEPT
         SELECT codcli FROM facturas
         WHERE  COALESCE(iva,0) = 18 ) AS t;
```

Trabajando con EXCEPT en lugar de NOT IN no es preciso preocuparse por los nulos en la clave ajena FACTURAS.codcli. Otra ventaja es que no aparecen en el resultado los clientes sin facturas. Además, suele suceder que las consultas así formuladas consiguen mejores tiempos de respuesta que las que utilizan NOT IN, quizá porque hay ciertas comprobaciones que se evitan.

4.10. Subconsultas correlacionadas

Una subconsulta correlacionada es una consulta anidada que contiene referencias a columnas de las tablas que se encuentran en el FROM de la consulta principal. Son lo que se denomina referencias externas.

Como ya se ha visto, las subconsultas dotan al lenguaje SQL de una gran potencia. Estas pueden utilizarse para hacer restricciones, tanto en la cláusula WHERE como en la cláusula HAVING, y también en la cláusula FROM. Hasta ahora, dichas subconsultas podían tratarse de modo independiente y, para comprender mejor el funcionamiento de la sentencia, se podía suponer que la subconsulta se ejecuta en primer lugar, y se sustituye ésta en la sentencia SELECT principal por su valor, como se muestra en el siguiente ejemplo:

```
-- facturas con descuento máximo
SELECT *
FROM   facturas
WHERE  dto = ( SELECT MAX(dto) FROM facturas );
```

en primer lugar se obtiene el descuento máximo de las facturas, se sustituye la subconsulta por este valor y, por último, se ejecuta la consulta principal.

4.10.1. Referencias externas

En ocasiones sucede que la subconsulta se debe recalcular para cada fila de la consulta principal, estando la subconsulta parametrizada mediante valores de columnas de la consulta principal. A este tipo de subconsultas se les llama subconsultas correlacionadas y a los parámetros de la subconsulta que pertenecen a la consulta principal se les llama referencias externas.

La siguiente sentencia obtiene los datos de las facturas que tienen descuento en todas sus líneas:

```
SELECT *
FROM   facturas AS f
WHERE  0 < ( SELECT MIN(COALESCE(l.dto,0))
             FROM   lineas_fac AS l
             WHERE  l.codfac = f.codfac );
```

La referencia externa es `f.codfac`, ya que es una columna de la consulta principal. En este caso, se puede imaginar que la consulta se ejecuta del siguiente modo. Se recorre, fila a fila, la tabla de las facturas. Para cada fila se ejecuta la subconsulta, sustituyendo `f.codfac` por el valor que tiene en la fila actual de la consulta principal. Es decir, para cada factura se obtiene el descuento mínimo en sus líneas. Si este descuento mínimo es mayor que cero, significa que la factura tiene descuento en todas sus líneas, por lo que se muestra en el resultado. Si no es así, la factura no se muestra. En cualquiera de los dos casos, se continua procesando la siguiente factura: se obtienen sus líneas y el descuento mínimo en ellas, etc.

4.10.2. Operadores `EXISTS, NOT EXISTS`

En un apartado anterior se han presentado los operadores que se pueden utilizar con las subconsultas para hacer restricciones en las cláusulas `WHERE` y `HAVING`. En aquel momento no se citó, intencionadamente, un operador, ya que éste se utiliza siempre con referencias externas: el operador `EXISTS`.

`EXISTS ( subconsulta )`
La subconsulta se evalúa para determinar si devuelve o no alguna fila. Si devuelve al menos una fila, se evalúa a verdadero. Si no devuelve ninguna fila, se evalúa a falso. La subconsulta puede tener referencias externas, que actuarán como constantes durante la evaluación de la subconsulta.

En la ejecución de la subconsulta, en cuanto se devuelve la primera fila, se devuelve verdadero, sin terminar de obtener el resto de las filas.

Puesto que el resultado de la subconsulta carece de interés (sólo importa si se devuelve o no alguna fila), se suelen escribir las consultas indicando una constante en la cláusula `SELECT` en lugar de `*` o cualquier columna:

```
-- facturas que en alguna línea no tiene dto
SELECT *
FROM   facturas AS f
WHERE  EXISTS ( SELECT 1
                FROM   lineas_fac AS l
                WHERE  l.codfac = f.codfac
                AND    COALESCE(dto,0)=0);
```

`NOT EXISTS ( subconsulta )`
La subconsulta se evalúa para determinar si devuelve o no alguna fila. Si devuelve al menos una fila, se evalúa a falso. Si no devuelve ninguna fila, se evalúa a verdadero. La subconsulta puede tener referencias externas, que actuarán como constantes durante la evaluación de la subconsulta.

En la ejecución de la subconsulta, en cuanto se devuelve la primera fila, se devuelve falso, sin terminar de obtener el resto de las filas.

Puesto que el resultado de la subconsulta carece de interés (sólo importa si se devuelve o no alguna fila), se suelen escribir las consultas indicando una constante en la cláusula `SELECT` en lugar de `*` o cualquier columna:

```
-- facturas que no tienen líneas sin descuento
SELECT *
FROM   facturas AS f
WHERE  NOT EXISTS ( SELECT 1
                    FROM   lineas_fac AS l
                    WHERE  l.codfac = f.codfac
                    AND    COALESCE(dto,0)=0);
```

4.10.3. Sentencias equivalentes

Algunos SGBD no son eficientes procesando consultas que tienen subconsultas anidadas con referencias externas, por lo que es muy conveniente saber encontrar sentencias equivalentes que no las utilicen, si es posible.

Por ejemplo, la siguiente sentencia también obtiene los datos de las facturas que tienen descuento en todas sus líneas. Utiliza una subconsulta en la cláusula `FROM` y no posee referencias externas.

```
SELECT *
FROM   facturas JOIN
       ( SELECT codfac
         FROM   lineas_fac
         GROUP BY codfac
         HAVING MIN(COALESCE(dto,0))>0 ) AS lf
       USING (codfac);
```

Una sentencia equivalente, que tampoco utiliza referencias externas, es la siguiente:

```
SELECT *
FROM   facturas
WHERE  codfac IN ( SELECT codfac
                   FROM   lineas_fac
                   GROUP BY codfac
                   HAVING MIN(COALESCE(dto,0))>0 );
```

4.10.4. Ejemplos

Ejemplo 4.16 *¿Cuántos clientes hay que en todas sus facturas han pagado 18 % de IVA?*

En la primera versión se van a utilizar operadores de conjuntos:

```
SELECT COUNT(*) AS clientes
FROM   (SELECT codcli FROM facturas
        WHERE iva = 18
        EXCEPT
        SELECT codcli FROM facturas
        WHERE COALESCE(iva,0) <> 18) AS t;
```

La siguiente sentencia no utiliza la subconsulta del `FROM`, pero seguramente será más cara porque hay que acceder a la tabla clientes:

```
SELECT COUNT(*) AS clientes
FROM   clientes
WHERE  codcli IN (SELECT codcli FROM facturas
                  WHERE iva = 18
                  EXCEPT
                  SELECT codcli FROM facturas
                  WHERE COALESCE(iva,0) <> 18);
```

La siguiente versión utiliza `NOT IN`, aunque ya se sabe que puede dar problemas cuando hay nulos:

```
SELECT COUNT(*) AS clientes
FROM   clientes
WHERE  codcli IN (SELECT codcli FROM facturas
                  WHERE  iva = 18)
AND    codcli NOT IN (SELECT codcli FROM facturas
                      WHERE  COALESCE(iva,0) <> 18
                      AND codcli IS NOT NULL);
```

La siguiente sentencia sigue una estrategia diferente: se ha pagado siempre el 18 % de IVA si el IVA máximo y el mínimo son ambos 18.

```
SELECT COUNT(*) AS clientes
FROM   clientes
WHERE  codcli IN (SELECT codcli FROM facturas
                  GROUP BY codcli
                  HAVING MAX(COALESCE(iva,0)) = 18
                  AND    MIN(COALESCE(iva,0)) = 18 );
```

Con la subconsulta en el `FROM` es posible evitar la visita de la tabla de los clientes:

```
SELECT COUNT(*) AS clientes
FROM   (SELECT codcli FROM facturas
        GROUP BY codcli
        HAVING MAX(COALESCE(iva,0)) = 18
        AND    MIN(COALESCE(iva,0)) = 18 ) AS t;
```

Ejemplo 4.17 *¿Cuántos pueblos hay en los que no tenemos clientes?*

Una versión con operadores de conjuntos es la siguiente:

```
SELECT COUNT(*) AS pueblos
FROM   (SELECT codpue FROM pueblos
        EXCEPT
        SELECT codpue FROM clientes) AS t;
```

Otra versión es la que utiliza `NOT IN`.

```
SELECT COUNT(*) AS pueblos
FROM   pueblos
WHERE  codpue NOT IN (SELECT codpue FROM clientes);
```

Ejemplo 4.18 *Para proponer ofertas especiales a los buenos clientes, se necesita un listado con los datos de aquellos que en los últimos quince meses (los últimos 450 días) han hecho siempre facturas por un importe superior a 400 €.*

Se puede pensar en obtener el resultado recorriendo, uno a uno, los clientes. Para cada cliente, comprobar, mediante una subconsulta, la restricción: que todas sus facturas de los últimos 450 días tengan un importe superior a 400 €. Ya que la subconsulta se ha de ejecutar para cada cliente, llevará una referencia externa.

La restricción que se ha de cumplir sobre todas las facturas de ese periodo se puede comprobar con `ALL` o con `NOT EXISTS`: o bien todas las facturas del

cliente (en el periodo) tienen un importe superior a 400 €, o bien no existen facturas de ese cliente (en el periodo) con un importe igual o inferior a 400 €.

Se debe tener en cuenta que con los dos operadores (ALL, NOT EXISTS) se obtendrán también en el resultado los clientes que no tienen ninguna factura, por lo que será preciso asegurarse de que los clientes seleccionados hayan comprado en alguna ocasión en dicho periodo.

A continuación se muestran las dos versiones de la consulta que utilizan las referencias externas tal y como se ha explicado.

```
SELECT c.codcli, c.nombre
FROM   clientes c
WHERE  400 < ALL ( SELECT SUM(l.cant*l.precio)
                   FROM   lineas_fac l JOIN facturas f
                          USING(codfac)
                   WHERE  f.fecha >= CURRENT_DATE - 450
                   AND    f.codcli = c.codcli  -- ref. externa
                   GROUP BY f.codfac )
AND    c.codcli IN ( SELECT f.codcli
                     FROM   facturas f
                     WHERE  f.fecha >= CURRENT_DATE - 450 )
ORDER BY c.nombre;
```

Nótese que con NOT EXISTS el predicado sobre el importe de las facturas es el único que debe aparecer negado.

```
SELECT c.codcli, c.nombre
FROM   clientes c
WHERE  NOT EXISTS ( SELECT 1
                    FROM   lineas_fac l JOIN facturas f
                           USING(codfac)
                    WHERE  f.fecha >= CURRENT_DATE - 450
                    AND    f.codcli = c.codcli  -- ref. externa
                    GROUP BY f.codfac
                    HAVING SUM(l.cant*l.precio) <= 400 )
AND    c.codcli IN ( SELECT f.codcli
                     FROM   facturas f
                     WHERE  f.fecha >= CURRENT_DATE - 450 )
ORDER BY c.nombre;
```

En la siguiente versión se evitan las referencias externas utilizando operadores de conjuntos. Obsérvese la subconsulta: del conjunto de los clientes que alguna vez han comprado en ese periodo con facturas de más de 400 €, se debe eliminar a aquellos que además han comprado alguna de 400 € o menos. Puesto que se utiliza el operador IN, no es necesaria la restricción adicional que comprueba que los clientes seleccionados hayan comprado alguna vez en el periodo: si están en la lista es porque lo han hecho.

```
SELECT c.codcli, c.nombre
FROM   clientes c
WHERE  c.codcli IN ( SELECT f.codcli
                     FROM   lineas_fac l JOIN facturas f
                            USING(codfac)
                     WHERE  f.fecha >= CURRENT_DATE - 450
                     GROUP BY f.codcli, f.codfac
                     HAVING SUM(l.cant*l.precio) > 400
                     EXCEPT
                     SELECT f.codcli
                     FROM   lineas_fac l JOIN facturas f
                            USING(codfac)
                     WHERE  f.fecha >= CURRENT_DATE - 450
                     GROUP BY f.codcli, f.codfac
                     HAVING SUM(l.cant*l.precio) <= 400 )
ORDER BY c.nombre;
```

Capítulo 5

Metodología de diseño de bases de datos

Introducción y objetivos

Una vez estudiado el modelo relacional de bases de datos, abordamos en esta segunda parte su diseño. El diseño de una base de datos debe realizarse siguiendo una metodología que garantice que se tienen en cuenta todos los requisitos de información y funcionales de la futura aplicación informática que la utilizará. En este capítulo se revisa el ciclo de vida de los sistemas de información ya que el diseño de la base de datos es una de sus etapas. A continuación se introduce brevemente la metodología de diseño que se abordará en detalle en los tres capítulos que siguen a éste.

Al finalizar este capítulo, el estudiante debe ser capaz de:

- Justificar la necesidad de utilizar metodologías en el diseño de bases de datos.
- Enumerar las etapas del ciclo de vida de un sistema de información y describir el objetivo de cada una de ellas.
- Describir las etapas del diseño de una base de datos.
- Justificar la necesidad de analizar no sólo los datos, sino también las transacciones, cuando se debe diseñar una base de datos.

5.1. Necesidad de metodologías de diseño

Cuando se trata de construir una base de datos sucede como cuando queremos que nos construyan una casa. Para la construcción de la casa no contratamos directamente a un constructor que la vaya haciendo sobre la marcha y como él quiera, sino que buscamos primero a un arquitecto que la diseñe en función de nuestras necesidades y contratamos al constructor después. El arquitecto, además de tener en cuenta nuestros requisitos, también tendrá en cuenta otros requisitos relativos a las estructuras, el sistema eléctrico o la seguridad.

Preocuparse por el diseño de las bases de datos es fundamental para la integridad de los datos. Si una base de datos está mal diseñada, los usuarios tendrán dificultades a la hora de acceder a los datos, las búsquedas podrán producir información errónea y podrán perderse datos o modificarse de manera incorrecta. Un mal diseño puede repercutir muy negativamente a la empresa propietaria de los datos. De hecho, si los datos de una base de datos van a influir en la gestión del negocio, si van a servir para tomar decisiones de la empresa, el diseño de la base de datos debe ser una verdadera preocupación.

El diseño de una base de datos se lleva a cabo en tres etapas: diseño conceptual, diseño lógico y diseño físico. Volviendo al símil con la construcción de una casa, el diseño conceptual y el lógico corresponden con la fase de elaboración de los planos arquitectónicos, mientras que la implementación física de la base de datos es la casa ya construida. Concretamente, diseño lógico describe el tamaño, la forma y los sistemas necesarios para la base de datos: contiene las necesidades en cuanto a información a almacenar y el modo en que se opera con ella. Después, se construye la implementación física del diseño lógico de la base de datos mediante el SGBD. Si pensamos en un sistema relacional, una vez creadas las tablas, establecidas las relaciones y los requisitos de integridad necesarios, la base de datos está finalizada. Después ya se pueden crear las aplicaciones que permitan interactuar con los datos de la base de datos. Con un buen diseño se puede garantizar que las aplicaciones proporcionarán la información oportuna y, sobre todo, la información correcta.

Hay ciertos factores que se consideran críticos en el diseño de bases de datos. Los que se citan a continuación son de gran importancia para afrontar con éxito el diseño de bases de datos.

- Trabajar interactivamente con los usuarios, tanto como sea posible.
- Utilizar una metodología estructurada durante todo el proceso de modelado de los datos.
- Emplear una metodología orientada a los datos (frente a una orientada a las funciones).
- Incluir en el modelado de los datos todo tipo de consideraciones estructurales, semánticas y de integridad.

- Utilizar diagramas para representar los datos siempre que sea posible.
- Mantener un diccionario de datos para complementar los diagramas.
- Estar dispuesto a repetir fases del diseño.

5.2. Ciclo de vida de los sistemas de información

Un *sistema de información* es el conjunto de recursos que permiten recoger, gestionar, controlar y difundir la información de toda una empresa u organización.

Desde los años setenta, los sistemas de bases de datos han ido reemplazando a los sistemas de ficheros en los sistemas de información de las empresas. Al mismo tiempo, se ha ido reconociendo la gran importancia que tienen los datos que éstas manejan, hasta convertirse en uno de sus recursos más importantes. Esto ha hecho que muchas empresas tengan departamentos que se encarguen de gestionar toda su información, que estará almacenada en una base de datos. Aparecen los papeles del *administrador de datos* y del *administrador de la base de datos*, que son las personas encargadas de supervisar y controlar todas las actividades relacionadas con los datos de la empresa y con el ciclo de vida de las aplicaciones de bases de datos, respectivamente.

Un sistema de información está formado por los siguientes componentes:

- La base de datos.
- El SGBD.
- Los programas de aplicación.
- Los dispositivos físicos (ordenadores, dispositivos de almacenamiento, etc.).
- El personal que utiliza y que desarrolla el sistema.

La base de datos es un componente fundamental de un sistema de información. El ciclo de vida de un sistema de información está ligado al ciclo de vida del sistema de base de datos sobre el que se apoya.

Las etapas del ciclo de vida de una sistema de información que se apoya sobre una base de datos son las siguientes:

1. Planificación del proyecto.
2. Definición del sistema.
3. Recolección y análisis de los requisitos.

4. Diseño de la base de datos.
5. Selección del SGBD.
6. Diseño de la aplicación.
7. Prototipado.
8. Implementación.
9. Conversión y carga de datos.
10. Prueba.
11. Mantenimiento.

Estas etapas no son estrictamente secuenciales. De hecho hay que repetir algunas de las etapas varias veces, haciendo lo que se conocen como *ciclos de realimentación*. Por ejemplo, los problemas que se encuentran en la etapa del diseño de la base de datos pueden requerir una recolección de requisitos adicional y su posterior análisis.

A continuación, se muestran las tareas más importantes que se realizan en cada etapa.

5.2.1. Planificación del proyecto

Esta etapa conlleva la planificación de cómo se pueden llevar a cabo las etapas del ciclo de vida de la manera más eficiente. Hay tres componentes principales: el trabajo que se ha de realizar, los recursos para llevarlo a cabo y el dinero para pagar por todo ello. Como apoyo a esta etapa, se necesitará un *esquema de datos* en donde se muestren las entidades principales de la empresa y sus relaciones, y en donde se identifiquen las principales áreas funcionales. En el esquema se tiene que mostrar también qué datos comparten las distintas áreas funcionales de la empresa.

La planificación de la base de datos también incluye el desarrollo de estándares que especifiquen cómo realizar la recolección de datos, cómo especificar su formato, qué documentación será necesaria y cómo se va a llevar a cabo el diseño y la implementación. El desarrollo y el mantenimiento de los estándares puede llevar bastante tiempo, pero si están bien diseñados, son una base para el personal informático en formación y para medir la calidad. Además, garantizan que el trabajo se ajusta a unos patrones, independientemente de las habilidades y la experiencia del diseñador. Por ejemplo, se puede establecer reglas sobre cómo dar nombres a los datos, lo que evitará redundancias e inconsistencias. Se deben documentar todos los aspectos legales sobre los datos y los establecidos por la empresa como, por ejemplo, qué datos deben tratarse de modo confidencial.

5.2.2. Definición del sistema

En esta etapa se especifica el ámbito y los límites de la aplicación de bases de datos, así como con qué otros sistemas interactúan. También hay que determinar quiénes son los usuarios y las áreas de aplicación.

5.2.3. Recolección y análisis de los requisitos

En esta etapa se recogen y analizan los requisitos de los usuarios y de las áreas funcionales de la empresa u organización. Esta información se puede recoger de varias formas:

- Entrevistando al personal de la empresa, concretamente, a aquellos que son considerados expertos en las áreas de interés.
- Observando el funcionamiento de la empresa.
- Examinando documentos, sobre todo aquellos que se utilizan para recoger o visualizar información.
- Utilizando cuestionarios para recoger información de grandes grupos de usuarios.
- Utilizando la experiencia adquirida en el diseño de sistemas similares.

La información recogida debe incluir las principales áreas funcionales y los grupos de usuarios, la documentación utilizada o generada por todos ellos, las transacciones que realizan y una lista priorizada de todos sus requisitos.

Esta etapa tiene como resultado un conjunto de documentos con las especificaciones de requisitos de los usuarios, en donde se describen las operaciones que se realizan en la empresa desde distintos puntos de vista.

La información recogida se debe estructurar utilizando *técnicas de especificación de requisitos*, como por ejemplo técnicas de análisis y diseño estructurado y diagramas de flujo de datos. También las herramientas CASE (*Computer-Aided Software Engineering*) pueden proporcionar una asistencia automatizada que garantice que los requisitos son completos y consistentes.

5.2.4. Diseño de la base de datos

Esta etapa consta de tres fases: diseño conceptual, diseño lógico y diseño físico de la base de datos. La primera fase consiste en la producción de un esquema conceptual de los datos, que es independiente de todas las consideraciones físicas. Este modelo se refina después en un esquema lógico eliminando las construcciones que no se pueden representar en el modelo de base de datos escogido (relacional, orientado a objetos, etc.). En la tercera fase, el esquema lógico se traduce en un esquema físico para el SGBD escogido. La fase de diseño físico debe tener en cuenta las estructuras de almacenamiento y los métodos de acceso necesarios para proporcionar un acceso eficiente a la base de datos en memoria secundaria.

Los objetivos del diseño de la base de datos son:

- Representar los datos que requieren las principales áreas funcionales y los usuarios, y representar las relaciones entre dichos datos.
- Proporcionar un modelo de los datos que soporte las transacciones que se vayan a realizar sobre los datos.
- Especificar un esquema que alcance las prestaciones requeridas para el sistema.

5.2.5. Selección del SGBD

Si no se dispone de un SGBD, o el que hay se encuentra obsoleto, se debe escoger un SGBD que sea adecuado para el sistema de información. Esta elección se debe hacer antes del diseño lógico.

5.2.6. Diseño de la aplicación

En esta etapa se diseñan los programas de aplicación que usarán y procesarán la base de datos. Esta etapa y el diseño de la base de datos, son paralelas. En la mayor parte de los casos no se puede finalizar el diseño de las aplicaciones hasta que se ha terminado con el diseño de la base de datos. Por otro lado, la base de datos existe para dar soporte a las aplicaciones, por lo que habrá una realimentación desde el diseño de las aplicaciones al diseño de la base de datos.

En esta etapa hay que asegurarse de que toda la funcionalidad especificada en los requisitos de usuario se encuentra en el diseño de la aplicación.

Además, habrá que diseñar las interfaces de usuario, aspecto muy importante que no se debe ignorar. El sistema debe ser fácil de aprender, fácil de usar, directo y estar dispuesto a tolerar ciertos fallos de los usuarios.

5.2.7. Prototipado

Esta etapa, que es opcional, es para construir prototipos de la aplicación que permitan a los diseñadores y a los usuarios probar el sistema. Un prototipo es un modelo de trabajo de las aplicaciones del sistema. El prototipo no tiene toda la funcionalidad del sistema final, pero es suficiente para que los usuarios puedan utilizar el sistema e identificar qué aspectos están bien y cuáles no son adecuados, además de poder sugerir mejoras o la inclusión de nuevos elementos. Este proceso permite que quienes diseñan e implementan el sistema sepan si han interpretado correctamente los requisitos de los usuarios. Otra ventaja de los prototipos es que se construyen rápidamente.

Esta etapa es imprescindible cuando el sistema que se va a implementar tiene un gran coste, alto riesgo o utiliza nuevas tecnologías.

5.2.8. Implementación

En esta etapa se crean las definiciones de la base de datos a nivel conceptual, externo e interno, así como los programas de aplicación. La implementación de la base de datos se realiza mediante las sentencias del lenguaje de definición de datos del SGBD escogido. Estas sentencias se utilizan para crear el esquema físico de la base de datos, los ficheros en donde se almacenarán los datos de la base de datos y las vistas de los usuarios.

Los programas de aplicación se implementan utilizando lenguajes de tercera o cuarta generación. Partes de estas aplicaciones son transacciones sobre la base de datos, que se implementan mediante el lenguaje de manejo de datos del SGBD. Las sentencias de este lenguaje se pueden embeber en un lenguaje de programación anfitrión como Visual Basic, Delphi, C, C++ o Java, entre otros. En esta etapa también se implementan los menús, los formularios para la introducción de datos y los informes de visualización de datos. Para ello, el SGBD puede disponer de lenguajes de cuarta generación que permiten el desarrollo rápido de aplicaciones mediante lenguajes de consultas no procedurales, generadores de informes, generadores de formularios, generadores de gráficos y generadores de aplicaciones.

En esta etapa también se implementan todos los controles de seguridad e integridad. Algunos de estos controles se pueden implementar mediante el lenguaje de definición de datos y otros puede que haya que implementarlos mediante utilidades del SGBD o mediante los programas de aplicación.

5.2.9. Conversión y carga de datos

Esta etapa es necesaria cuando se está reemplazando un sistema antiguo por uno nuevo. Los datos se cargan desde el sistema viejo al nuevo directamente o, si es necesario, se convierten al formato que requiera el nuevo SGBD y luego se cargan. Si es posible, los programas de aplicación del sistema antiguo también se convierten para que se puedan utilizar en el sistema nuevo.

5.2.10. Prueba

En esta etapa se prueba y valida el sistema con los requisitos especificados por los usuarios. Para ello, se debe diseñar una batería de test con datos reales, que se deben llevar a cabo de manera metódica y rigurosa. Es importante darse cuenta de que la fase de prueba no sirve para demostrar que no hay fallos, sirve para encontrarlos. Si la fase de prueba se lleva a cabo correctamente, descubrirá los errores en los programas de aplicación y en la estructura de la base de datos. Además, demostrará que los programas parecen trabajar tal y como se especificaba en los requisitos y que las prestaciones deseadas parecen obtenerse. Por último, en las pruebas se podrá hacer una medida de la fiabilidad y la calidad del *software* desarrollado.

5.2.11. Mantenimiento

Una vez que el sistema está completamente implementado y probado, se pone en marcha. Se dice que el sistema está ahora en la fase de mantenimiento, en la que se llevan a cabo las siguientes tareas:

- Monitorización de las prestaciones del sistema. Si las prestaciones caen por debajo de un determinado nivel, puede ser necesario reorganizar la base de datos.
- Mantenimiento y actualización del sistema. Cuando sea necesario, los nuevos requisitos que vayan surgiendo se incorporarán al sistema, siguiendo de nuevo las etapas del ciclo de vida que se acaban de presentar.

5.3. Diseño de bases de datos

En este apartado se describen con más detalle los objetivos de cada una de las etapas del diseño de bases de datos: diseño conceptual, diseño lógico y diseño físico. La metodología a seguir en cada una de estas etapas se describe con detalle en capítulos posteriores.

5.3.1. Diseño conceptual

En esta etapa se debe construir un esquema de la información que se usa en la empresa, independientemente de cualquier consideración física. A este esquema se le denomina *esquema conceptual*. Al construir el esquema, los diseñadores descubren la semántica (significado) de los datos de la empresa: encuentran entidades, atributos y relaciones. El objetivo es comprender:

- La perspectiva que cada usuario tiene de los datos.
- La naturaleza de los datos, independientemente de su representación física.
- El uso de los datos a través de las áreas funcionales.

El esquema conceptual se puede utilizar para que el diseñador transmita a la empresa lo que ha entendido sobre la información que ésta maneja. Para ello, ambas partes deben estar familiarizadas con la notación utilizada en el esquema. La más popular es la notación del modelo entidad-relación, que se describe en el capítulo dedicado al diseño conceptual.

El esquema conceptual se construye utilizando la información que se encuentra en la especificación de los requisitos de usuario. El diseño conceptual es completamente independiente de los aspectos de implementación, como puede ser el SGBD que se vaya a usar, los programas de aplicación, los lenguajes de programación, el *hardware* disponible o cualquier otra consideración física. Durante todo el proceso de desarrollo del esquema conceptual éste se prueba y se valida con los requisitos de los usuarios. El esquema conceptual es una fuente de información para el diseño lógico de la base de datos.

5.3.2. Diseño lógico

El diseño lógico es el proceso de construir un esquema de la información que utiliza la empresa, basándose en un modelo de base de datos específico, independiente del SGBD concreto que se vaya a utilizar y de cualquier otra consideración física.

En esta etapa, se transforma el esquema conceptual en un esquema lógico que utilizará las estructuras de datos del modelo de base de datos en el que se basa el SGBD que se vaya a utilizar, como pueden ser: el modelo relacional, el modelo de red, el modelo jerárquico o el modelo orientado a objetos. Conforme se va desarrollando el esquema lógico, éste se va probando y validando con los requisitos de usuario.

La *normalización* es una técnica que se utiliza para comprobar la validez de los esquemas lógicos basados en el modelo relacional, ya que asegura que las tablas obtenidas no tienen datos redundantes. Esta técnica se presenta en el capítulo dedicado al diseño lógico de bases de datos.

El esquema lógico es una fuente de información para el diseño físico. Además, juega un papel importante durante la etapa de mantenimiento del sistema, ya que permite que los futuros cambios que se realicen sobre los programas de aplicación o sobre los datos, se representen correctamente en la base de datos.

Tanto el diseño conceptual, como el diseño lógico, son procesos iterativos, tienen un punto de inicio y se van refinando continuamente. Ambos se deben ver como un proceso de aprendizaje en el que el diseñador va comprendiendo el funcionamiento de la empresa y el significado de los datos que maneja. El diseño conceptual y el diseño lógico son etapas clave para conseguir un sistema que funcione después correctamente. Si el esquema no es una representación fiel de la empresa, será difícil, sino imposible, definir todas las vistas de usuario (esquemas externos) o mantener la integridad de la base de datos. También puede ser difícil definir la implementación física o mantener unas prestaciones aceptables del sistema. Además, hay que tener en cuenta que la capacidad de ajustarse a futuros cambios es un sello que identifica a los buenos diseños. Por todo esto, es fundamental dedicar el tiempo y las energías necesarias para producir el mejor esquema que sea posible.

5.3.3. Diseño físico

El diseño físico es el proceso de producir la descripción de la implementación de la base de datos en memoria secundaria: determinar las estructuras de almacenamiento y los métodos de acceso que garanticen un acceso eficiente a los datos.

Para llevar a cabo esta etapa, se debe haber decidido cuál es el SGBD que se va a utilizar, ya que el esquema físico se adapta a él. Entre el diseño físico y el diseño lógico hay una realimentación, ya que algunas de las decisiones que se tomen durante el diseño físico para mejorar las prestaciones pueden afectar a la estructura del esquema lógico.

En general, el propósito del diseño físico es describir cómo se va a implementar físicamente el esquema lógico obtenido en la fase anterior. Concretamente, en el modelo relacional, esto consiste en:

- Obtener un conjunto de tablas y determinar las restricciones que se debe cumplir sobre ellas.
- Determinar las estructuras de almacenamiento y los métodos de acceso que se van a utilizar para conseguir unas prestaciones óptimas.
- Diseñar el modelo de seguridad del sistema.

5.4. Diseño de transacciones

Cuando se diseñan las aplicaciones, se deben diseñar también las transacciones que éstas contienen y que son las encargadas de trabajar sobre la base de datos. Una transacción es un conjunto de acciones llevadas a cabo por un usuario o un programa de aplicación, que acceden o cambian el contenido de la base de datos. Las transacciones representan eventos del mundo real, como dar de alta un nuevo cliente, registrar una factura o dar de baja un artículo que ya no está a la venta. Estas transacciones se deben realizar sobre la base de datos para que ésta siga siendo un fiel reflejo de la realidad.

Una transacción puede estar compuesta por varias operaciones sobre la base de datos, como registrar una factura, que requiere insertar datos en varias tablas. Sin embargo, desde el punto de vista del usuario, estas operaciones conforman una sola tarea. Desde el punto de vista del SGBD, una transacción lleva a la base de datos de un estado consistente a otro estado consistente. El SGBD garantiza la consistencia de la base de datos incluso si se produce algún fallo, y también garantiza que una vez se ha finalizado una transacción, los cambios realizados por ésta quedan permanentemente en la base de datos, no se pueden perder ni deshacer (a menos que se realice otra transacción que compense el efecto de la primera). Si la transacción no se puede finalizar por cualquier motivo, el SGBD garantiza que los cambios realizados por esta transacción son deshechos.

El objetivo del diseño de las transacciones es definir y documentar las características de alto nivel de las mismas que requiere el sistema. Esta tarea se debe llevar a cabo al principio del proceso de diseño para garantizar que el esquema lógico es capaz de soportar todas las transacciones necesarias. Las características que se debe recoger de cada transacción son las siguientes:

- Datos que utiliza la transacción.
- Características funcionales de la transacción.
- Salida de la transacción.
- Importancia para los usuarios.
- Frecuencia de utilización.

Hay tres tipos de transacciones:

- En las *transacciones de recuperación* se accede a los datos para visualizarlos en la pantalla a modo de informe.
- En las *transacciones de actualización* se insertan, borran o actualizan datos de la base de datos.
- En las *transacciones mixtas* se mezclan operaciones de recuperación de datos y de actualización.

El diseño de las transacciones utiliza la información dada en las especificaciones de requisitos de usuario.

5.5. Herramientas CASE

Cuando se hace la planificación de la base de datos (la primera etapa del ciclo de vida de las aplicaciones de bases de datos), también se puede escoger una herramienta CASE que permita llevar a cabo el resto de tareas del modo más eficiente y efectivo posible. Una herramienta CASE suele incluir:

- Un diccionario de datos para almacenar información sobre los datos de la aplicación de bases de datos.
- Herramientas de diseño para dar apoyo al análisis de datos.
- Herramientas que permitan desarrollar el modelo de datos corporativo, así como los esquemas conceptual y lógico.
- Herramientas para desarrollar los prototipos de las aplicaciones.

El uso de las herramientas CASE puede mejorar la productividad en el desarrollo de una aplicación de bases de datos, tanto desde el punto de vista de la eficiencia durante su desarrollo, como de la efectividad del sistema desarrollado. La eficiencia se refiere al coste, tanto en tiempo como en dinero, de desarrollar la aplicación. La efectividad se refiere al grado en que el sistema satisface las necesidades de los usuarios. Para obtener una buena productividad, subir el nivel de efectividad puede ser más importante que aumentar la eficiencia.

Capítulo 6

Diseño conceptual

Introducción y objetivos

El primer paso en el diseño de una base de datos es la producción del esquema conceptual. En este capítulo se presenta una metodología para producir estos esquemas, denominada *entidad-relación*.

Al finalizar este capítulo, el estudiante debe ser capaz de:

- Captar una realidad determinada, correspondiente a unos requisitos de usuario, y plasmarla en un esquema conceptual mediante un diagrama entidad-relación.
- Interpretar un esquema conceptual dado, extrayendo de él los requisitos de datos de los usuarios que se hayan reflejado.

6.1. Modelo entidad-relación

El diseño conceptual parte de las especificaciones de requisitos de los usuarios y su resultado es el esquema conceptual de la base de datos. Una opción para recoger los requisitos consiste en examinar los diagramas de flujo de datos, que se pueden haber producido previamente, para identificar cada una de las áreas funcionales. La otra opción consiste en entrevistar a los usuarios, examinar los procedimientos, los informes y los formularios, y también observar el funcionamiento de la empresa.

Un esquema conceptual es una descripción de alto nivel de la estructura de la base de datos, independientemente del SGBD que se vaya a utilizar para manipularla. Para especificar los esquemas conceptuales se utilizan modelos conceptuales. Los modelos conceptuales se utilizan para representar la realidad a un alto nivel de abstracción. Mediante los modelos conceptuales se puede construir una descripción de la realidad fácil de entender. En el diseño de bases de datos se usan, en primer lugar, los modelos conceptuales para lograr

una descripción de alto nivel de la realidad, y luego se transforma el esquema conceptual en un esquema lógico (diseño lógico).

Los modelos conceptuales deben ser buenas herramientas para representar la realidad, por lo que deben poseer las siguientes cualidades:

- *Expresividad*: deben tener suficientes conceptos para expresar perfectamente la realidad.
- *Simplicidad*: deben ser simples para que los esquemas sean fáciles de entender.
- *Minimalidad*: cada concepto debe tener un significado distinto.
- *Formalidad*: todos los conceptos deben tener una interpretación única, precisa y bien definida.

En general, un modelo no es capaz de expresar todas las propiedades de una realidad determinada, por lo que hay que añadir afirmaciones que complementen el esquema.

El modelo entidad-relación es el modelo conceptual más utilizado para el diseño conceptual de bases de datos. Fue introducido por Peter Chen en 1976. El modelo entidad-relación está formado por un conjunto de conceptos que permiten describir la realidad mediante representaciones gráficas y lingüísticas. Estos conceptos se muestran en la figura 6.1:

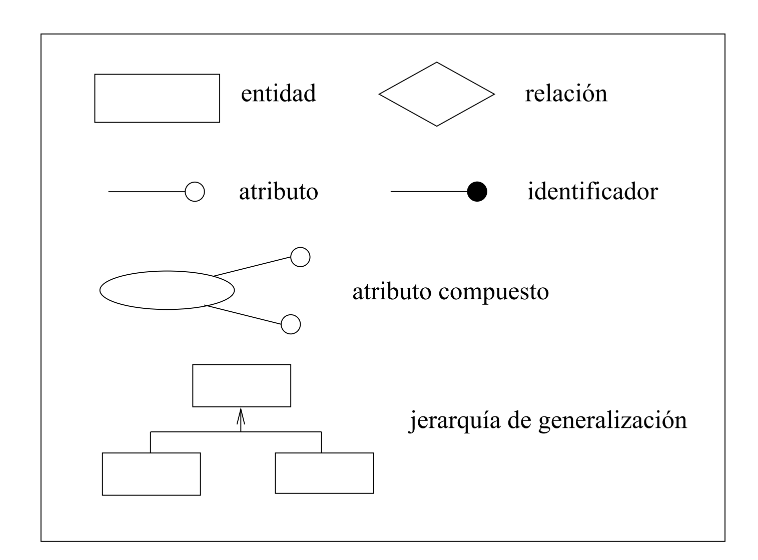


Figura 6.1: Conceptos del modelo entidad-relación.

Originalmente, el modelo entidad-relación sólo incluía los conceptos de entidad, relación y atributo. Más tarde, se añadieron otros conceptos, como los atributos compuestos y las jerarquías de generalización, en lo que se ha denominado modelo entidad-relación extendido.

Las tareas a realizar en el diseño conceptual son las siguientes:

1. Identificar las entidades.
2. Identificar las relaciones.
3. Identificar los atributos y asociarlos a entidades y relaciones.
4. Determinar los dominios de los atributos.
5. Determinar los identificadores.
6. Determinar las jerarquías de generalización (si las hay).
7. Dibujar el diagrama entidad-relación.
8. Revisar el esquema conceptual local con el usuario.

6.1.1. Entidades

En primer lugar, hay que definir los principales conceptos que interesan al usuario. Estos conceptos serán las entidades. Una forma de identificar las entidades es examinar las especificaciones de requisitos de usuario. En estas especificaciones se buscan los nombres o los sintagmas nominales que se mencionan (por ejemplo: código del cliente, nombre del cliente, número de la factura, fecha de la factura, IVA de la factura). También se buscan conceptos importantes como personas, lugares o conceptos abstractos, excluyendo aquellos nombres que sólo son propiedades de otros objetos. Por ejemplo, se pueden agrupar el código del cliente y el nombre del cliente en una entidad denominada *cliente*, y agrupar el número de la factura, la fecha de la factura y el IVA de la factura en otra entidad denominada *factura*.

Otra forma de identificar las entidades es buscar aquellos conceptos que existen por sí mismos. Por ejemplo, *vendedor* es una entidad porque los vendedores existen, sepamos o no sus nombres, direcciones y teléfonos. Siempre que sea posible, el usuario debe colaborar en la identificación de las entidades.

A veces, es difícil identificar las entidades por la forma en que aparecen en las especificaciones de requisitos. Los usuarios, a veces, hablan utilizando ejemplos o analogías. En lugar de hablar de vendedores en general, hablan de personas concretas, o bien, hablan de los puestos que ocupan esas personas.

Para complicarlo aún más, los usuarios usan, muchas veces, sinónimos y homónimos. Dos palabras son sinónimos cuando tienen el mismo significado. Los homónimos ocurren cuando la misma palabra puede tener distintos significados dependiendo del contexto.

No siempre es obvio saber si un concepto es una entidad, una relación o un atributo. El análisis es subjetivo, por lo que distintos diseñadores pueden hacer distintas interpretaciones, aunque todas igualmente válidas. Todo depende de la opinión y la experiencia de cada uno. Los diseñadores de bases de datos

deben tener una visión selectiva y clasificar las cosas que observan dentro del contexto de la empresa u organización. A partir de unas especificaciones de usuario es posible que no se pueda deducir un conjunto único de entidades, pero después de varias iteraciones del proceso de análisis, se llegará a obtener un conjunto de entidades que sean adecuadas para el sistema que se ha de construir.

Conforme se van identificando las entidades, se les dan nombres que tengan un significado y que sean obvias para el usuario. Los nombres de las entidades y sus descripciones se anotan en el diccionario de datos. Cuando sea posible, se debe anotar también el número aproximado de ocurrencias de cada entidad. Si una entidad se conoce por varios nombres, éstos se deben anotar en el diccionario de datos como alias o sinónimos. En el modelo entidad-relación, las entidades se representan mediante un rectángulo que posee dentro el nombre de la entidad.

Ejemplo 6.1 *Entidades.*

CIUDADES y ASIGNATURAS se han representado como entidades porque de ellas se requiere almacenar información: nombre de la ciudad, provincia en la que se encuentra, número de habitantes, nombre de la asignatura, créditos teóricos y prácticos, titulación a la que pertenece, etc.

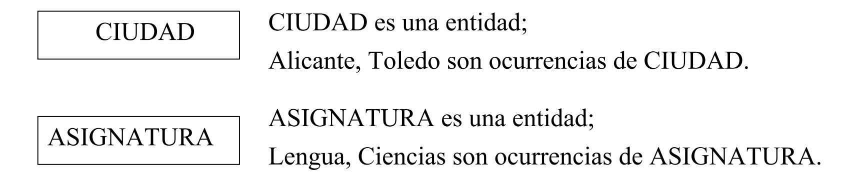


Figura 6.2: Ejemplos de dos entidades y de ocurrencias de las mismas.

6.1.2. Relaciones

Una vez definidas las entidades, se debe definir las relaciones existentes entre ellas. Del mismo modo que para identificar las entidades se buscaban nombres en las especificaciones de requisitos, para identificar las relaciones se suelen buscar las expresiones verbales. Por ejemplo: ciudad donde ha nacido el estudiante y ciudades en que ha residido; cada director tiene a su cargo a un conjunto de empleados. Si las especificaciones de requisitos reflejan estas relaciones es porque son importantes para la empresa y, por lo tanto, se deben reflejar en el esquema conceptual. La mayoría de las relaciones son binarias (entre dos entidades), pero también puede haber relaciones en las que participen más de dos entidades, así como relaciones recursivas.

Es muy importante repasar las especificaciones para comprobar que todas las relaciones, explícitas o implícitas, se han encontrado. Si se tienen pocas

entidades, se puede comprobar por parejas si hay alguna relación entre ellas. De todos modos, las relaciones que no se identifican ahora se suelen encontrar cuando se valida el esquema con las transacciones que debe soportar.

Una vez identificadas todas las relaciones, hay que determinar la cardinalidad mínima y máxima con la que participa cada entidad en cada una de ellas. De este modo, el esquema representa de una manera más explícita la semántica de las relaciones. La cardinalidad es un tipo de restricción que se utiliza para comprobar y mantener la calidad de los datos.

La cardinalidad mínima indica si la participación de la entidad en la relación es opcional (se indica con 0) o si es obligatoria (se indica con 1). Que sea obligatoria implica que todas las ocurrencias de la entidad deberán relacionarse con, al menos, una ocurrencia de la entidad que se encuentra al otro lado de la relación. La cardinalidad máxima indica si cada ocurrencia de la entidad sólo puede relacionarse con una ocurrencia de la entidad del otro lado de la relación (se indica con 1), o si puede relacionarse con varias a la vez (se indica con n).

Conforme se van identificando las relaciones, se les van asignando nombres que tengan significado para el usuario. En el diccionario de datos se anotan los nombres de las relaciones, su descripción y las restricciones que existen sobre ellas.

Ejemplo 6.2 *Tipos de relaciones.*

Las entidades se relacionan entre ellas o consigo mismas, lo cual se representa en el esquema conceptual mediante líneas y un rombo en donde se da nombre a la relación. En la línea se expresa la cardinalidad con la que cada entidad participa en la relación mediante dos componentes entre paréntesis.

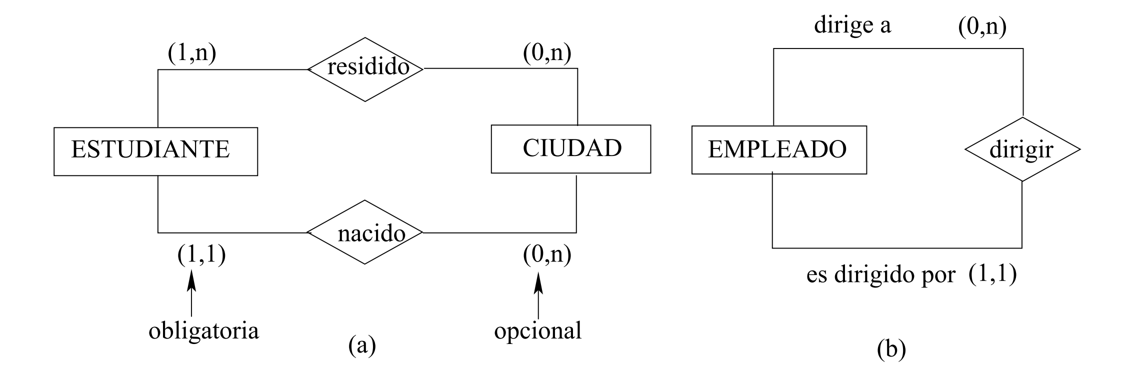


Figura 6.3: Ejemplos de relaciones.

Los esquemas de la figura 6.3 corresponden a los siguientes requisitos:

(a) De cada estudiante se sabe la ciudad en donde ha nacido (será una y sólo una) y también las ciudades en donde ha residido (al menos aquella en la que reside en la actualidad).

(b) Cada empleado es dirigido por otro empleado (obligatoriamente por uno y sólo por uno), y un empleado puede ser director de varios empleados (o no serlo de ninguno).

6.1.3. Atributos

El siguiente paso consiste en identificar los atributos y asociarlos con las entidades y las relaciones en función de su significado. Al igual que ha procedido con las entidades, para identificar los atributos se buscan nombres en las especificaciones de requisitos. Son atributos los nombres que identifican propiedades, cualidades, identificadores o características de entidades o de relaciones.

Lo más sencillo es preguntarse, para cada entidad y cada relación, qué información se quiere saber de ellas. La respuesta a esta pregunta se debe encontrar en las especificaciones de requisitos. Pero, en ocasiones, será necesario preguntar a los usuarios para que aclaren los requisitos. Desgraciadamente, los usuarios pueden dar respuestas a esta pregunta que también contengan otros conceptos, por lo que hay que considerar sus respuestas con mucho cuidado.

Al identificar los atributos, hay que tener en cuenta si son simples o compuestos. Por ejemplo, el atributo *dirección* puede ser simple, teniendo la dirección completa como un solo valor: 'San Rafael 45, Almazora'; o puede ser un atributo compuesto, formado por la *calle* ('San Rafael'), el *número* ('45') y la *población* ('Almazora'). El escoger entre atributo simple o compuesto depende de los requisitos del usuario. Si el usuario no necesita acceder a cada uno de los componentes de la dirección por separado, se puede representar como un atributo simple. Pero si el usuario quiere acceder a los componentes de forma individual, entonces se debe representar como un atributo compuesto.

En el esquema conceptual se debe reflejar la cardinalidad mínima y máxima de cada atributo, ya sea simple o compuesto. La cardinalidad mínima indica si el atributo es opcional (se expresa con 0) o si es obligatorio (se expresa con 1). La cardinalidad máxima indica si el atributo tiene, como mucho, un solo valor (se indica con 1) o si puede tener varios valores, es decir, si es multievaluado (se indica con n). Puesto que el valor más usual en la cardinalidad de los atributos es «(1,1)» (tienen un valor y sólo uno), ésta se omite para estos casos, siendo el valor por defecto.

En esta fase también se debe identificar los atributos derivados o calculados, que son aquellos cuyo valor se puede calcular a partir de los valores de otros atributos. Por ejemplo, el número de estudiantes matriculados, la edad de los estudiantes o el número de ciudades en que residen los estudiantes. Algunos diseñadores no representan los atributos derivados en los esquemas conceptuales. Si se hace, se debe indicar claramente que el atributo es derivado y a partir de qué atributos se obtiene su valor. El momento en que hay que considerar los atributos derivados es en el diseño físico.

Cuando se están identificando los atributos, se puede descubrir alguna entidad que no se ha identificado previamente, por lo que hay que volver al

principio introduciendo esta entidad y viendo si se relaciona con otras entidades.

Es muy útil elaborar una lista con los atributos que aparecen en los requisitos e ir eliminándolos de la lista conforme se vayan asociando a una entidad o relación. De este modo, uno se puede asegurar de que cada atributo se asocia a una sola entidad o relación, y que cuando la lista se ha acabado, se han asociado todos los atributos.

Hay que tener mucha precaución cuando parece que un mismo atributo se debe asociar a varias entidades. Esto puede ser por una de las siguientes causas:

- Se han identificado varias entidades, como *director*, *supervisor* y *administrativo*, cuando, de hecho, pueden representarse como una sola entidad denominada *empleados*. En este caso, se puede escoger entre introducir una jerarquía de generalización (se presentan más adelante), o dejar las entidades que representan cada uno de los puestos que ocupan los empleados.

- Se ha identificado una relación entre entidades. En este caso, se debe asociar el atributo a una sola de las entidades y hay que asegurarse de que la relación ya se había identificado previamente. Si no es así, se debe actualizar el esquema y el diccionario, para recoger la nueva relación.

Conforme se van identificando los atributos, se les asignan nombres que tengan significado para el usuario. De cada atributo se debe anotar la siguiente información en el diccionario:

- Nombre y descripción del atributo.
- Alias o sinónimos por los que se conoce al atributo.
- Tipo de dato y longitud.
- Valores por defecto del atributo (si se especifican).
- Si el atributo es compuesto, especificar qué atributos simples lo forman y describirlos como se indica en esta lista.
- Si el atributo es derivado, indicar cómo se calcula su valor.

Ejemplo 6.3 *Atributos simples.*

De los estudiantes del diagrama de la figura 6.4 se quiere conocer el nombre, el DNI y la carrera que están estudiando. Conocer la ciudad de nacimiento es, en este caso, opcional (cardinalidad $(0,1)$) y, si se conoce, se conocerá también la fecha de nacimiento. Es por ello que este último atributo está en la relación y no en la entidad estudiante: va ligado a conocer o no la ciudad

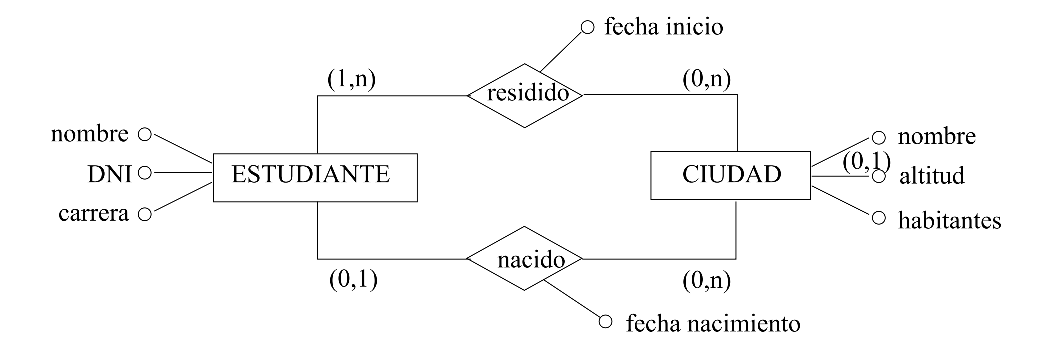


Figura 6.4: Ejemplo de atributos simples.

de nacimiento. Además, de los estudiantes también interesan las ciudades en donde han residido y la fecha en que han comenzado a hacerlo en cada una de ellas. De las ciudades, interesa su nombre y su número de habitantes, y si es posible, su altitud. El que las ciudades participen en ambas relaciones con cardinalidad $(0, n)$ significa que hay ciudades en donde puede que no haya nacido ningún estudiante o que hayan nacido varios, y que hay ciudades en donde puede que no haya residido ningún estudiante o que hayan residido varios.

Ejemplo 6.4 *Atributos compuestos.*

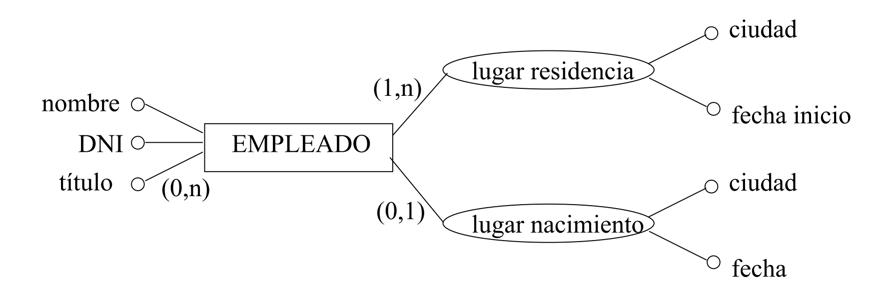


Figura 6.5: Ejemplo de atributos compuestos.

El diagrama de la figura 6.5 corresponde a unos requisitos muy similares a los del ejemplo anterior: datos de empleados, lugar de nacimiento y lugares de residencia. En este caso, las ciudades no se han considerado como entidad porque de ellas no hay que conocer otras propiedades aparte de su nombre. En este ejemplo aparecen atributos compuestos y atributos multievaluados (con cardinalidad máxima n). La interpretación del esquema es la siguiente: de los empleados interesa su nombre y su DNI, además del título o títulos que tienen, si es el caso (puede haber empleados sin titulación). Si se conoce el lugar de nacimiento, interesa el nombre de la ciudad y la fecha. Además, interesa conocer los lugares en que ha residido y, para cada uno de ellos, el nombre de la ciudad y la fecha de inicio de la residencia.

6.1.4. Dominios

En este paso se deben determinar los dominios de los atributos. El dominio de un atributo es el conjunto de valores que puede tomar el atributo. Por ejemplo, el dominio de los DNI son las tiras de nueve caracteres en donde los ocho primeros son dígitos numéricos y el último es un carácter de control que se obtiene al aplicar un determinado algoritmo; el dominio de los códigos postales en España son cadenas de cinco dígitos, correspondiendo los dos primeros a un número de provincia válido.

Un esquema conceptual está completo si incluye los dominios de todos sus atributos, es decir, los valores permitidos para cada atributo, su tamaño y su formato. También se puede incluir información adicional sobre los dominios como, por ejemplo, las operaciones que se pueden realizar sobre cada atributo, qué atributos pueden compararse entre sí o qué atributos pueden combinarse con otros. Aunque sería muy interesante que el sistema final respetara todas estas indicaciones sobre los dominios, esto es todavía una línea abierta de investigación. Toda la información sobre los dominios se debe anotar también en el diccionario de datos.

6.1.5. Identificadores

Cada entidad tiene al menos un identificador. En este paso, se trata de encontrar todos los identificadores de cada una de las entidades. Los identificadores pueden ser simples o compuestos. De cada entidad se escogerá uno de los identificadores como clave primaria en la fase del diseño lógico. Todos los identificadores de las entidades se deben anotar en el diccionario de datos.

Cuando se determinan los identificadores es fácil darse cuenta de si una entidad es fuerte o débil. Si una entidad tiene al menos un identificador, es *fuerte* (otras denominaciones son *padre*, *propietaria* o *dominante*). Si una entidad no tiene atributos que le sirvan de identificador, es *débil* (otras denominaciones son *hijo*, *dependiente* o *subordinada*).

Ejemplo 6.5 *Identificadores de entidades fuertes.*

El diagrama de la figura 6.6 muestra cómo se identifican estudiantes y ciudades. No conviene olvidar que estamos trabajando ante unos supuestos requisitos. En este caso, se sabe que los estudiantes se identifican de modo único por su DNI, y las ciudades por su nombre.

Ejemplo 6.6 *Identificadores de entidades débiles.*

En el diagrama de la figura 6.7 se muestra una entidad débil: la de los empleados. Cada empleado se identifica por su número de empleado dentro de su departamento. Nótese que los departamentos tienen dos posibles formas de identificarse: bien mediante su número o bien mediante su nombre. Por lo tanto

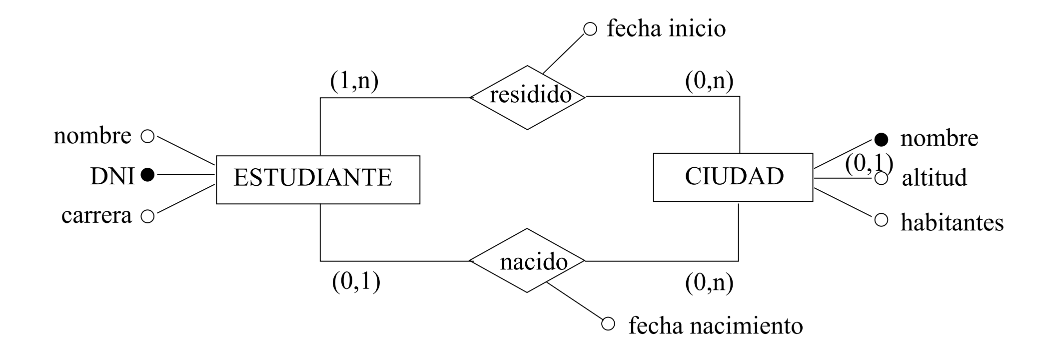


Figura 6.6: Ejemplo de identificador de entidades fuertes.

hay también dos maneras de identificar a los empleados: por la combinación de su número de empleado y el número de su departamento, o bien, por la combinación de su número de empleado y el nombre de su departamento.

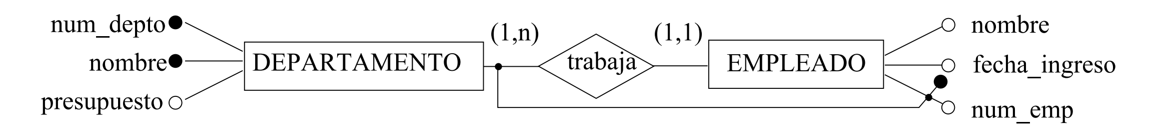


Figura 6.7: Ejemplo de identificador de entidad débil.

6.1.6. Jerarquías de generalización

En este paso hay que observar las entidades que se han identificado hasta el momento. Hay que ver si es necesario reflejar las diferencias entre distintas ocurrencias de una entidad, con lo que surgirán nuevas subentidades de esta entidad genérica; o bien, si hay entidades que tienen características en común y que realmente son subentidades de una nueva entidad genérica.

En cada jerarquía hay que determinar la cardinalidad mínima y máxima. La cardinalidad mínima expresa si cada ocurrencia de la entidad está obligada o no a estar clasificada en alguna subentidad. Si está obligada se dice que la jerarquía es total y si no lo está, se dice que es parcial. La cardinalidad máxima expresa si cada ocurrencia de la entidad se clasifica sólo como una subentidad o si puede estar clasificada como varias. Si lo está sólo en una, se dice que es exclusiva, si no es superpuesta. Esta cardinalidad se expresa, bien con letras o bien con números: $(p/t, e/s) \equiv (0/1, 1/n)$.

Ejemplo 6.7 *Jerarquía de generalización.*

La figura 6.8 muestra una jerarquía que clasifica las pólizas de una compañía de seguros. Todas ellas tienen un número que las identifica, una fecha de inicio

y una fecha de finalización. Además, si una póliza es de un seguro de vida, se conoce la información de sus beneficiarios (puede ser más de uno). Si la póliza es de un seguro de automóvil, se conoce la matrícula del mismo. Puesto que un atomóvil sólo puede tener una póliza, su matrícula es también un identificador de la misma. Por último, si la póliza es de un seguro de vivienda, se conoce el domicilio de la vivienda asegurada.

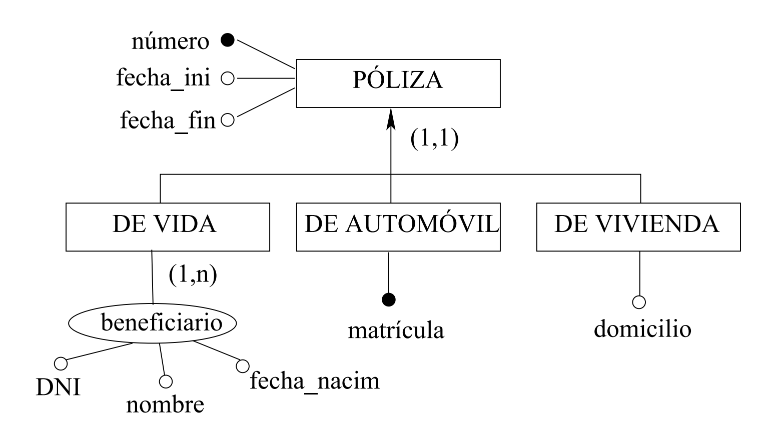


Figura 6.8: Ejemplo de jerarquía de generalización.

6.1.7. Diagrama entidad-relación

Una vez identificados todos los conceptos (entidades, atributos, relaciones, etc.), se debe dibujar el diagrama entidad-relación correspondiente a cada una de las vistas de los usuarios. Se obtienen así los esquemas conceptuales locales.

Antes de dar por finalizada la fase del diseño conceptual, se debe revisar cada esquema conceptual local con los usuarios. Estos esquemas están formados por cada diagrama entidad-relación y toda la documentación que describe cada esquema. Si se encuentra alguna anomalía, hay que corregirla haciendo los cambios oportunos, por lo que posiblemente haya que repetir alguno de los pasos anteriores. Este proceso debe repetirse hasta que se esté seguro de que cada esquema conceptual es una fiel representación de la parte de la empresa que se está tratando de modelar.

6.2. Recomendaciones

En este apartado se dan algunas recomendaciones para dibujar los esquemas conceptuales y se muestran los errores más comunes que se cometen en los diagramas.

- Dos entidades no se pueden conectar directamente con una línea (ver figura 6.9). La forma de conectar entidades es mediante relaciones.



Figura 6.9: No es correcto conectar entidades directamente (excepto si forman parte de una jerarquía).

- No puede haber conexiones entre dos relaciones (ver figura 6.10).

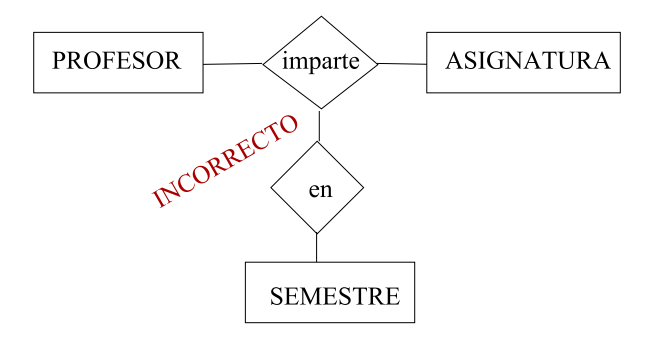


Figura 6.10: No es correcto conectar relaciones.

- Los atributos se asocian a entidades y a relaciones, pero no se asocian a las líneas que las conectan (ver figura 6.11).

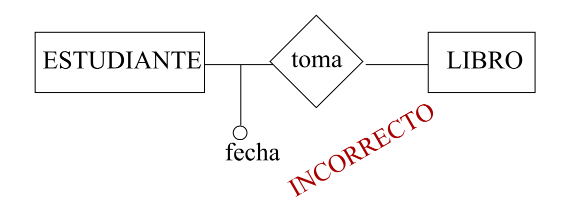


Figura 6.11: No es correcto colocar atributos fuera de entidades y relaciones.

- Cuando una entidad participa en una relación, se debe indicar siempre la cardinalidad con la que participa «(0/1, 1/n)».

- Un atributo es una propiedad de una entidad o de una relación. Cada atributo se dibuja sólo una vez en el esquema.
- Puede haber nombres de atributos iguales en distintas entidades siempre que tengan significados diferentes. Por ejemplo, la entidad `ESTUDIANTE` tiene un atributo `nombre` y la entidad `UNIVERSIDAD` también puede tener un atributo `nombre`, teniendo, cada uno de estos atributos, un significado diferente.
- Los atributos simples se representan mediante círculos pequeños conectados directamente a la entidad o la relación con una línea en la que se especifica la cardinalidad. La cardinalidad por defecto es «(1,1)». Junto a cada círculo se especifica el nombre del atributo (el nombre que debe ser significativo). Los atributos que no forman parte de un atributo compuesto deben unirse con líneas independientes a la entidad (no es correcto hacerlo como se muestra en la figura 6.12).



Figura 6.12: No es correcto usar una misma línea para los atributos.

- Los atributos compuestos se representan mediante un óvalo, especificando su nombre en el interior, y tendrán uno o varios atributos simples conectados directamente a él mediante una línea. El atributo compuesto estará conectado a la entidad o la relación mediante una línea en la que se especificará la cardinalidad. La cardinalidad por defecto es «(1,1)».
- La cardinalidad de un atributo no expresa su rango de valores (ver figura 6.13). El rango de valores posibles es el dominio sobre el que se define el atributo. La cardinalidad es el número de valores distintos que puede tener el atributo a la vez.

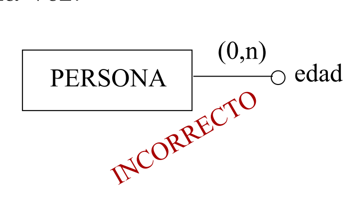


Figura 6.13: No es correcto usar la cardinalidad para expresar el rango de valores.

- Si un atributo tiene un número fijo de posibles valores, éstos no se dibujan como componentes de un atributo compuesto (ver figura 6.14). Al especificar el dominio del atributo será cuando se especifiquen los posibles valores.

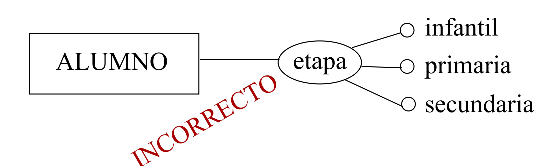


Figura 6.14: No es correcto usar los atributos compuestos para expresar rangos de valores.

- Todas las entidades deben tener, al menos, un identificador. Cada identificador se representa mediante un círculo relleno de color. Los atributos con cardinalidad máxima n no pueden ser identificadores. En el diseño conceptual se deben tener en cuenta todos los posibles identificadores y dibujarlos.[1]

- Cuando un identificador está formado por varios atributos, éstos no tendrán su círculo coloreado (ver figura 6.15). Los atributos que forman el identificador se deben dejar sin colorear, se conectan mediante una línea y es al final de la misma donde se dibuja un círculo coloreado, indicando así que la combinación de todos los atributos conectados forma un identificador de la entidad.

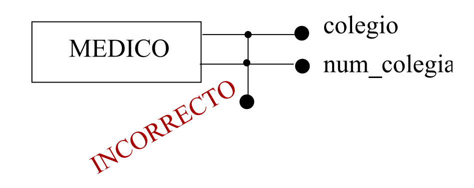


Figura 6.15: No es correcto colorear los atributos simples que forman un identificador compuesto.

- Las relaciones no tienen identificadores (ver figura 6.16).

- Una entidad débil es aquella que depende de otra para identificarse. Su participación en la relación con la entidad de la que depende será siempre «(1,1)». El identificador de la entidad débil estará formado por uno o varios de sus atributos, en combinación con el identificador de la entidad de la que depende. Esto se expresa conectando con una línea dichos

[1]Todos ellos serán claves candidatas en la etapa del diseño lógico: uno terminará siendo la clave primaria (`PRIMARY KEY`) y el resto serán claves alternativas (`UNIQUE`).

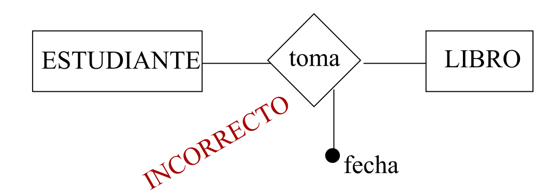


Figura 6.16: No es correcto poner identificadores a las relaciones.

atributos, y la línea que conecta a la otra entidad con la relación de dependencia. Al final de la línea se dibujará un círculo coloreado, expresando así el identificador. Los demás atributos que lo forman no deben colorearse (ver figura 6.17).

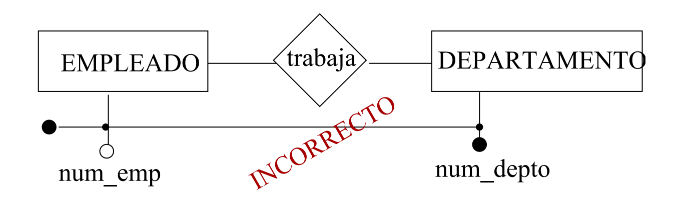


Figura 6.17: No es correcto colorear los atributos simples que forman parte de un identificador compuesto.

6.3. Ejemplos

Ejemplo 6.8 *Asociación de cines.*

«La asociación de cines de una ciudad quiere crear un servicio telefónico en el que se pueda hacer cualquier tipo de consulta sobre las películas que se están proyectando actualmente. Algunos ejemplos de consultas son los siguientes: en qué cines hacen una determinada película y el horario de los pases, qué películas de dibujos animados se están proyectando y dónde, qué películas hay en un determinado cine, etc. La aplicación informática que se va a implementar necesitará de una base de datos relacional que contenga toda esta información. Como primer paso, en este ejercicio se pide realizar el esquema conceptual.

En concreto, para cada cine se debe dar el título de la película y el horario de los pases, además del nombre del director de la misma, el nombre de hasta tres de sus protagonistas, el género (comedia, intriga, etc.) y la clasificación (todos los públicos, a partir de 13 años, a partir de 18 años, etc.).

Para cada cine también se almacenará la calle y número donde está, el teléfono y los distintos precios según el día (día del espectador, día del jubilado, festivos y vísperas, etc.). Hay que tener en cuenta que algunos cines tienen

varias salas en las que se pasan distintas películas y también que en un mismo cine se pueden pasar películas distintas en diferentes pases.»

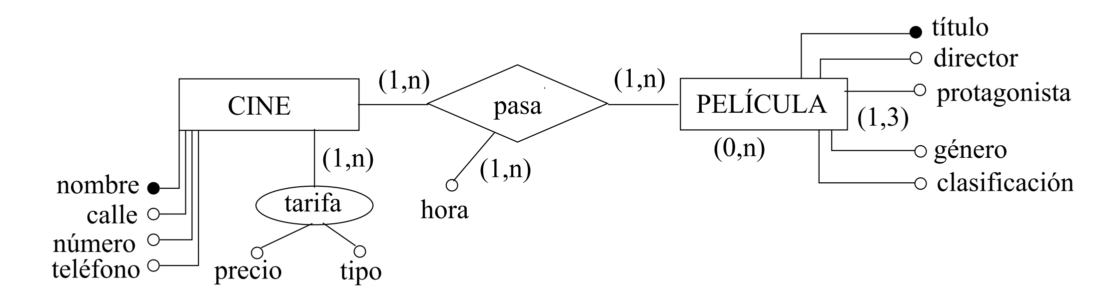


Figura 6.18: Esquema conceptual para el caso de la asociación de cines.

A partir de los requisitos especificados se ha obtenido el esquema conceptual de la figura 6.18. Se han identificado dos entidades: los cines y las películas. Son atributos de los cines su nombre, su dirección, su número de teléfono y la tarifa de precios (cada tipo de tarifa tiene un precio). Las películas tienen como atributos el título, el director, los nombres de hasta tres de sus protagonistas, el género y la clasificación.

La relación entre cines y películas se establece cuando éstos las incluyen en sus pases. Una película se puede pasar en varios horarios, y es por eso que se han incluido éstos en la relación

La siguiente tabla muestra las características de los atributos del esquema:

Atributo	**Tipo**	**Dominio**	**Ejemplo**
nombre	cadena		`Neocine Castellón`
calle	cadena		`Paseo Buenavista`
número	cadena		`s/n`
teléfono	cadena		`964 280 121`
tarifa.precio	moneda	> 0	`4,50`
tarifa.tipo	cadena		`Día del espectador`
hora	hora		`20:15`
título	cadena		`El niño con el pijama de rayas`
director	cadena		`Mark Herman`
protagonista	cadena		`David Thewlis`
género	cadena		`drama`
clasificación	cadena		`7 años`

Ejemplo 6.9 *Catálogo de un portal web.*

«Se desea incorporar un catálogo a un portal web y como primer paso, en este ejercicio se va a obtener el esquema conceptual de la base de datos que le dará soporte.

El catálogo se va a organizar como una lista jerárquica de temas. Cada tema final de la jerarquía tendrá un conjunto de enlaces a páginas web recomendadas. Por ejemplo, un tema podría ser `PostgreSQL`. Dentro de la jerarquía, éste podría ser un subtema (hijo) del tema `Sistemas de gestión de bases de datos`. El tema `MySQL` podría ser otro subtema de este último.

De cada tema final hay varias páginas web recomendadas. En el tema `PostgreSQL` una página podría ser `www.postgresql.org` y otra página podría ser la web donde están colgados estos apuntes. De cada página se guarda la URL y el título.

Para cada página se almacena una prioridad en cada tema en que se recomienda. Esta prioridad sirve para ordenarlas al mostrar los resultados de las búsquedas en el catálogo de temas. Por ejemplo, la página `www.postgresql.org` tendría una prioridad mayor que la de los apuntes que tienes en tus manos.

Cada tema tiene una serie de palabras clave asociadas, cada una con un número que permite ordenarlas según su importancia dentro del tema. Por ejemplo, el tema `PostgreSQL` podría tener las palabras clave `(1) relacional`, `(2) multiusuario` y `(3) libre`.

También se quiere guardar información sobre las consultas que se han realizado sobre cada tema del catálogo. Cada vez que se consulte un tema se guardará la IP de la máquina desde la que se ha accedido y la fecha y hora de la consulta.

Algunas páginas web son evaluadas por voluntarios. La calificación que otorgan es: `****` , `***`, `**` o `*`. Se debe almacenar información sobre los voluntarios (nombre y correo electrónico) y las evaluaciones que han hecho de cada página (calificación y fecha en que se ha valorado). Una misma página puede ser evaluada por distintos voluntarios y, ya que las páginas van cambiando su estructura y contenidos, pueden ser valoradas en más de una ocasión por un mismo voluntario. En el caso de repetir una evaluación de una misma página por un mismo voluntario, sólo interesa almacenar la última evaluación realizada (la más reciente).»

A partir de estos requisitos, se ha obtenido el esquema conceptual de la figura 6.19.

Se han identificado tres entidades: los temas del catálogo, las páginas web a las que apuntan los temas y los voluntarios que califican las páginas. Se han considerado atributos del tema su nombre, las palabras clave con su importancia (atributo compuesto con múltiples valores) y las consultas que se van realizando (IP e instante de tiempo).

La jerarquía de temas del catálogo se ha representado mediante una relación de la entidad de los temas consigo misma, de manera que algunas ocurrencias

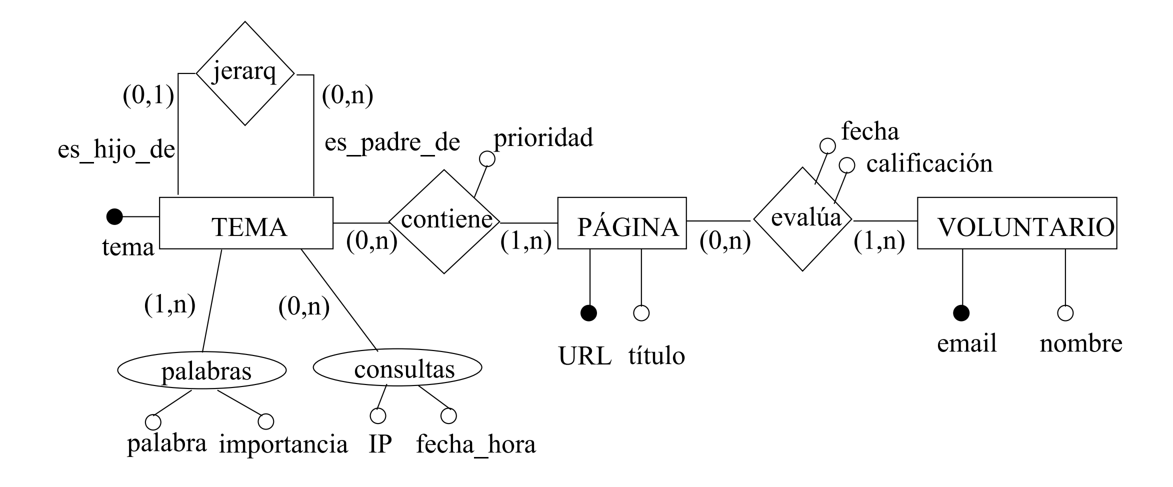


Figura 6.19: Esquema conceptual para el caso del catálogo web.

de esta entidad están relacionadas con otras ocurrencias de la misma. Cuando se establece una de estas relaciones, es importante etiquetar los caminos. Así se tiene que cada tema hijo, lo es sólo de un tema, y si es padre, puede serlo de varios temas.

Otra entidad identificada es la de las páginas web. De cada página se tiene la URL y el título, y puede ser apuntada por varios temas de la jerarquía.

La tercera entidad es la correspondiente a los voluntarios que califican las páginas. Cada voluntario tiene una dirección de correo electrónico (`email`) y su nombre. La relación entre voluntarios y páginas se establece cada vez que un voluntario califica una página. Los posibles valores del atributo calificación son: `****` , `***`, `**` o `*`.

En la tabla que aparece a continuación se muestran algunas características de los atributos. Nos hemos permitido la libertad de especificar tipos como `IP` o `URL`, ya que éstos tienen especificaciones conocidas y bien definidas. Las longitudes de las cadenas no se han especificado ya que en los requisitos del ejercicio no se ha proporcionado información al respecto.

Atributo	Tipo	Dominio	Ejemplo
tema	cadena		`PostgreSQL`
palabra	cadena		`relacional`
importancia	entero	> 0	`2`
IP	IP		`164.12.123.65`
fecha_hora	instante		`11/10/2008 13:23:10`
prioridad	entero	> 0	`5`
URL	URL		`www.postgresql.org`
título	cadena		`The world's most advanced open source database`
email	email		`persona@servicio.com`
nombre	cadena		`Mafalda Goreiro`
fecha	fecha	fecha actual	`11/10/2008`
calificación	cadena	$* * **, * * *, **, *$	`****`

Capítulo 7

Diseño lógico relacional

Introducción y objetivos

Una vez realizado el diseño conceptual, y obtenido el esquema correspondiente mediante un diagrama entidad-relación, se debe proceder con la etapa del diseño lógico. En esta etapa se debe decidir el modelo lógico de base de datos que se va a utilizar para llevar a cabo la implementación. Puesto que el modelo relacional es el modelo lógico de bases de datos más extendido, en este capítulo se presenta la metodología de diseño para este modelo.

Al finalizar este capítulo, el estudiante debe ser capaz de:

- Obtener un conjunto de tablas a partir de un esquema conceptual (expresado mediante un diagrama entidad-relación) y de las especificaciones adicionales expresadas en el diccionario de datos.
- Establecer para cada tabla: la clave primaria, las claves alternativas, las claves ajenas y las reglas de integridad para las mismas.
- Establecer las restricciones y reglas de negocio que se deben hacer sobre las tablas y sobre sus columnas.
- Obtener un diagrama entidad-relación a partir de un conjunto de tablas.

7.1. Esquema lógico

El diseño lógico es el proceso de construir un esquema de la información que utiliza la empresa, basándose en un modelo de base de datos específico e independiente del SGBD concreto que se vaya a utilizar, así como de cualquier otra consideración física. Mientras que el objetivo fundamental del diseño conceptual es la compleción y expresividad del esquema conceptual, el objetivo del diseño lógico es obtener una representación que use, del modo más eficiente posible, los recursos que el modelo de SGBD posee para estructurar los datos y para modelar las restricciones

En esta etapa, se transforma el esquema conceptual obtenido en la etapa anterior del diseño, en un esquema lógico que utilizará las estructuras de datos del modelo de base de datos en el que se basa el SGBD que se vaya a utilizar. Los modelos de bases de datos más extendidos son el modelo relacional, el modelo de red y el modelo jerárquico. El modelo orientado a objetos es también muy popular, existiendo SGBD objeto-relacionales que implementan el modelo relacional e incorporan características de la orientación a objetos.

El esquema lógico es una fuente de información para el diseño físico. Además, juega un papel importante durante la etapa de mantenimiento del sistema, ya que permite que los futuros cambios que se realicen sobre los programas de aplicación o sobre los datos, se representen correctamente en la base de datos.

Tanto el diseño conceptual, como el diseño lógico, son procesos iterativos, tienen un punto de inicio y se van refinando continuamente. Ambos se deben ver como un proceso de aprendizaje en el que el diseñador va comprendiendo el funcionamiento de la empresa y el significado de los datos que maneja. El diseño conceptual y el diseño lógico son etapas clave para conseguir un sistema que funcione correctamente. Si la base de datos no es una representación fiel de la empresa, será difícil, si no imposible, definir todas las vistas de los usuarios (los esquemas externos), o mantener la integridad de la misma. También puede ser difícil definir la implementación física o mantener unas prestaciones aceptables del sistema. Además, hay que tener en cuenta que la capacidad de ajustarse a futuros cambios es un sello que identifica a los buenos diseños de bases de datos. Por todo esto, es fundamental dedicar el tiempo y las energías necesarias para producir el mejor esquema posible.

La estructura de datos del modelo relacional es la relación (capítulo 2), a la que coloquialmente denominamos tabla, término utilizado en la implementación de este modelo por parte del lenguaje SQL (capítulo 4).

El objetivo de esta etapa es obtener el esquema lógico, que estará formado por las tablas de la base de datos en tercera forma normal,[1] a partir de la especificación realizada en la etapa del diseño conceptual. Una vez obtenidas las tablas, se considerará la posibilidad de modificar el esquema de la base de datos para conseguir una mayor eficiencia.

No se debe olvidar que, si en durante la etapa del diseño lógico se detecta alguna carencia o error en la etapa anterior (diseño conceptual), se debe subsanar dicho error en el esquema conceptual, y se debe generar una nueva versión de la documentación que se ha producido en dicha etapa.

Para cada tabla del esquema lógico se debe especificar:

- Nombre y descripción de la información que almacena. Es conveniente indicar si corresponde a una entidad, una relación o un atributo.
- Para cada columna, indicar: nombre, tipo de datos (puede ser un tipo de SQL), si admite nulos, el valor por defecto (si lo tiene) y el rango de valores (mediante un predicado en SQL).

[1] La tercera forma normal se presenta en el apartado 7.2.5, que trata la normalización.

- Indicar la clave primaria y si se ha de generar automáticamente.
- Indicar las claves alternativas.
- Indicar las claves ajenas y sus reglas de comportamiento ante el borrado y la modificación de la clave primaria a la que referencian.
- Si alguna columna es un dato derivado (su valor se calcula a partir de otros datos de la base de datos) indicar cómo se obtiene su valor.
- Especificar las restricciones a nivel de fila de cada tabla, si las hay. Estas restricciones son aquellas que involucran a una o varias columnas dentro de una misma fila.
- Especificar otras restricciones no expresadas antes (serán aquellas que involucran a varias filas de una misma tabla o a filas de varias tablas a la vez).
- Especificar las reglas de negocio, que serán aquellas acciones que se deba llevar a cabo de forma automática como consecuencia de actualizaciones que se realicen sobre la base de datos.
- Introducir tablas de referencia para establecer listas de valores para las columnas que las necesiten.

Una vez obtenido el esquema de la base de datos en tercera forma normal, y teniendo en cuenta los requisitos en cuanto a transacciones, volumen de datos y prestaciones deseadas, se puede realizar ciertos cambios que ayuden a conseguir una mayor eficiencia en el acceso a la base de datos:

- Introducir redundancias desnormalizando algunas tablas o añadiendo datos derivados.
- Partir tablas horizontalmente (por casos) o verticalmente (por columnas).

7.2. Metodología de diseño

En este apartado se presentan los pasos a seguir para obtener un conjunto de tablas a partir del esquema conceptual. A cada tabla se le dará un nombre, y el nombre de sus atributos aparecerá, a continuación, entre paréntesis. El atributo o atributos que forman la clave primaria se subrayarán. Las claves ajenas, mecanismo que se utiliza para representar las relaciones entre entidades en el modelo relacional, se especificarán aparte indicando la tabla a la que hacen referencia.

7.2.1. Entidades fuertes

En el esquema lógico se debe crear una tabla para cada entidad fuerte, incluyendo todos sus atributos simples con cardinalidad máxima 1. De los atributos compuestos con cardinalidad máxima 1 incluir sólo sus componentes.

Cada atributo con cardinalidad máxima n se incluirá como una tabla dentro de la tabla correspondiente a la entidad. Si el atributo es simple, la tabla interna tendrá una sola columna; si el atributo es compuesto, la tabla interna tendrá tantas columnas como componentes tenga éste.

Cada uno de los identificadores de la entidad será una clave candidata. De entre las claves candidatas hay que escoger la clave primaria; el resto serán claves alternativas. Para escoger la clave primaria entre las claves candidatas se puede seguir las siguientes indicaciones:

- Escoger la clave candidata que tenga menos atributos.
- Escoger la clave candidata cuyos valores no tengan probabilidad de cambiar en el futuro.
- Escoger la clave candidata cuyos valores no tengan probabilidad de perder la unicidad en el futuro.
- Escoger la clave candidata con el mínimo número de caracteres (si es de tipo cadena).
- Escoger la clave candidata más fácil de utilizar desde el punto de vista de los usuarios.

Ejemplo 7.1 *Entidad fuerte con atributos.*

El diagrama de la figura 7.1 contiene los datos de interés de los libros de una biblioteca: título (formado por un título principal y el subtítulo), ISBN, editorial, autores, idioma en que está escrito y ediciones (cada edición tiene un número y se ha publicado en un año).

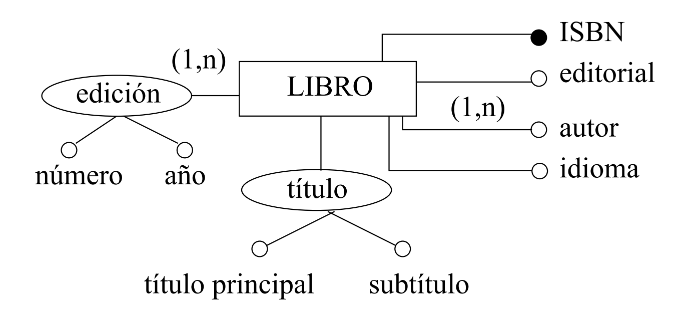


Figura 7.1: Entidad con atributos.

El esquema lógico correspondiente es el siguiente:

```
LIBRO( isbn, título_principal, subtítulo, editorial,
AUTOR( autor ), idioma, EDICIÓN( número, año ) )
```

7.2.2. Entidades débiles

En el esquema lógico se debe crear una tabla para cada entidad débil, teniendo en cuenta todos sus atributos tal y como se ha hecho con las entidades fuertes. Una entidad débil participa en una relación con la entidad fuerte de la que depende y la cardinalidad con la que participa será siempre «(1,1)»: cada ocurrencia de la entidad débil se relaciona con una y sólo una ocurrencia de la entidad fuerte, de la que necesita para identificarse. Por el hecho de participar de este modo en la relación y por ser débil, a la tabla que le corresponde se le debe añadir una clave ajena a la tabla de la entidad fuerte de la que depende. Para ello, se incluye la clave primaria de la tabla que representa a la entidad fuerte (padre) en la nueva tabla creada para la entidad débil. A continuación, se debe determinar la clave primaria de la nueva tabla.

Ejemplo 7.2 *Entidad débil con atributos.*

El diagrama de la figura 7.2 corresponde al ejemplo 6.6 del capítulo 6.

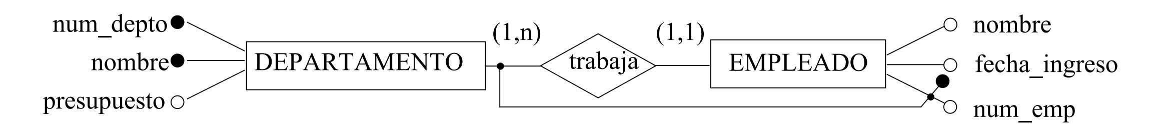


Figura 7.2: Ejemplo de identificador de entidad débil.

El esquema lógico correspondiente es el siguiente:

```
DEPARTAMENTO(num_depto, nombre, presupuesto)
DEPARTAMENTO.nombre es una clave alternativa
EMPLEADO(num_emp, num_depto, nombre, fecha_ingreso)
EMPLEADO.num_depto es una clave ajena a DEPARTAMENTO
```

7.2.3. Relaciones binarias

Una relación binaria es aquella en la que participan dos entidades, o bien una sola entidad cuyas ocurrencias se relacionan entre ellas (autorrelación). En los diagramas entidad-relación, para cada entidad, se especifica la cardinalidad con la que participa en cada relación. Según sean las cardinalidades máximas, las relaciones binarias se clasifican como se especifica a continuación:

- *Uno a uno*: ambas entidades participan con cardinalidad máxima 1. Si una participa de forma opcional y la otra lo hace de manera obligatoria, esta última es considerada la entidad hija, mientras que la primera es la entidad madre.
- *Uno a muchos*: una entidad participa con cardinalidad máxima 1 (será la entidad hija) mientras que la otra lo hace con cardinalidad máxima n (será la entidad madre).
- *Muchos a muchos*: ambas entidades participan con cardinalidad máxima n.

En función del tipo de relación, hay distintas posibilidades para representarlas en el esquema lógico.

Relaciones binarias uno a uno

Antes de transformar las relaciones uno a uno, es preciso revisarlas, ya que es posible que se hayan identificado dos entidades que representen el mismo concepto pero con nombres diferentes (sinónimos). Si así fuera, ambas entidades deben integrarse en una sola, y después debe obtenerse la tabla correspondiente.

Hay dos formas distintas de representar, en el esquema lógico, una relación binaria uno a uno entre entidades fuertes. Una vez obtenidas las tablas correspondientes a las entidades participantes en la relación las opciones son:

(a) Incluir en una de las tablas (sólo en una de ellas) una clave ajena a la otra tabla. Esta clave ajena será, a su vez, una clave alternativa, ya que cada ocurrencia de un lado sólo puede relacionarse con una ocurrencia del otro lado y viceversa. Además, se deben incluir en la misma tabla los atributos de la relación.

    La clave ajena aceptará nulos o no, en función de la cardinalidad mínima con la que participe la entidad correspondiente en la relación: si es 0, la participación es opcional por lo que debe aceptar nulos; si es 1, la participación es obligatoria y no debe aceptarlos. Los atributos de la relación que se han incluido en la tabla sólo aceptarán nulos si son opcionales, o bien, cuando la clave ajena deba aceptar nulos (participación opcional).

(b) Crear una nueva tabla para almacenar las ocurrencias de la relación. Esta tabla contendrá una clave ajena a cada una de las tablas correspondientes a las entidades participantes, además de incluir los atributos de la relación. Ninguna de las claves ajenas aceptará nulos, ya que la tabla almacena ocurrencias de la relación. Además, ambas claves ajenas serán claves candidatas: se escogerá una de ellas como clave primaria y la otra quedará como clave alternativa.

Si la relación corresponde a una entidad débil con la entidad fuerte de la que depende, lo único que se debe hacer es añadir los atributos de la relación (si los tiene), a la tabla de la entidad débil, puesto que ésta ya contiene la clave ajena a la tabla de la entidad fuerte, que además de ayudarle a identificarse (será una clave candidata), expresa la relación. Cuando se tiene una relación binaria uno a uno entre una entidad débil y una fuerte, puede ser conveniente plantearse la posibilidad de integrar las dos entidades en una sola, como si se tratara de sinónimos.

Ejemplo 7.3 *Relación uno a uno.*

El diagrama de la figura 7.3 contiene información de los empleados (código y nombre), de los vehículos que éstos conducen (matrícula y modelo) y desde cuando lo hacen.

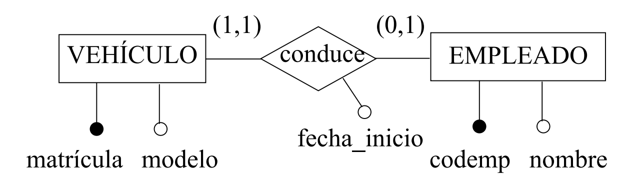


Figura 7.3: Relación de uno a uno.

A continuación, se muestran tres posibles esquemas lógicos correspondientes a este diagrama:

(a.1) Puesto que la entidad de los vehículos participa de forma obligatoria en la relación, puede considerarse entidad hija (todas sus ocurrencias están relacionadas con algún empleado), introduciéndose en ella la relación:

`EMPLEADO(`<u>`codemp`</u>`, nombre)`
`VEHÍCULO(`<u>`matrícula`</u>`, modelo, codemp, fecha_inicio)`
`VEHÍCULO.codemp` es una clave ajena a `EMPLEADO,` no acepta nulos
`VEHÍCULO.codemp` es también una clave alternativa

(a.2) Aunque la entidad de los vehículos es la entidad hija, al ser una relación uno a uno, también es posible incluir la relación en la entidad de los

empleados. Esto puede ser conveniente cuando se sabe que los accesos de una tabla a la otra se van a hacer siempre en la misma dirección, de EMPLEADO a VEHÍCULO:

> VEHÍCULO(<u>matrícula</u>, modelo)
> EMPLEADO(<u>codemp</u>, nombre, matrícula, fecha_inicio)
> EMPLEADO.matrícula es una clave ajena a VEHÍCULO, acepta nulos
> EMPLEADO.matrícula es también una clave alternativa
> EMPLEADO.matrícula, EMPLEADO.fecha_inicio son ambas nulas o no nulas a la vez

Nótese que, por el hecho de participar de manera opcional en la relación, la clave ajena y el atributo de la relación deben aceptar nulos, y que ambos deben ser nulos o no nulos a la vez. Esta restricción se puede expresar sin ambigüedad en forma de predicado SQL:

```
(EMPLEADO.matrícula IS NULL AND EMPLEADO.fecha_inicio
IS NULL)
OR (EMPLEADO.matrícula IS NOT NULL AND EMPLEADO.fecha_inicio
IS NOT NULL)
```

(b) Otro modo de representar la relación es mediante una tabla aparte:

> EMPLEADO(<u>codemp</u>, nombre)
> VEHÍCULO(<u>matrícula</u>, modelo)
> CONDUCE(<u>matrícula</u>, codemp, fecha_inicio)
> CONDUCE.matrícula es una clave ajena a VEHÍCULO, no acepta nulos
> CONDUCE.codemp es una clave ajena a EMPLEADO, no acepta nulos
> CONDUCE.codemp es también una clave alternativa

Nótese que ninguna de las claves ajenas acepta nulos, aún habiendo una entidad que participa de manera opcional. Esto es así porque la tabla CONDUCE almacena ocurrencias de una relación, no de una entidad: si la relación no se da para algún empleado, éste no aparece en la tabla.

Escoger una u otra opción para representar cada relación uno a uno dependerá, en gran medida, de cómo se va a acceder a las tablas y del número de ocurrencias de las entidades que van a participar en la relación. Se tratará siempre de favorecer los accesos más frecuentes y que requieran un tiempo de respuesta menor.

Por ejemplo, en el esquema (a.1) dar de alta un vehículo conlleva ejecutar una sola sentencia INSERT en la tabla VEHÍCULO, mientras que hacerlo en los esquemas (a.2) y (b) conlleva ejecutar dos sentencias (un INSERT y un UPDATE,

o dos `INSERT`). Sin embargo, en estos dos últimos esquemas, un recorrido completo de la tabla `VEHÍCULO` para obtener la matrícula y el modelo es más rápido puesto que cada fila almacena menos datos. Por otra parte, mantener la restricción de que todo vehículo debe estar relacionado con algún empleado (con la fecha de inicio), es trivial en el esquema (a.1) exigiendo que ambos atributos no acepten nulos, mientras que hacerlo en los otros dos esquemas requiere el uso de transacciones.

En resumen, cada esquema será más conveniente para ciertos tipos de accesos, por lo que se tratará de favorecer aquellos que sean críticos.

Relaciones binarias uno a muchos

Cuando la relación entre dos entidades fuertes es de uno a muchos, sigue habiendo dos modos de representarla en el esquema lógico: mediante una clave ajena o mediante una tabla aparte; aunque el modo de hacerlo varía respecto a las relaciones de uno a uno, tal y como se muestra a continuación:

(a) Incluir en la tabla hija (aquella cuya entidad participa con cardinalidad máxima 1) una clave ajena a la tabla madre, junto con los atributos de la relación. La clave ajena aceptará nulos o no, en función de la cardinalidad mínima con la que participe la entidad hija en la relación: si es 0, la participación es opcional por lo que debe aceptar nulos; si es 1, la participación es obligatoria y no debe aceptarlos. Los atributos de la relación que se han incluido en la tabla sólo aceptarán nulos si son opcionales, o bien cuando la clave ajena deba aceptar nulos (participación opcional).

(b) Crear una nueva tabla para almacenar las ocurrencias de la relación. Esta tabla contendrá una clave ajena a cada una de las tablas correspondientes a las entidades participantes, además de incluir los atributos de la relación. Ninguna de las claves ajenas aceptará nulos, ya que la tabla almacena ocurrencias de la relación. La clave primaria será la clave ajena a la tabla correspondiente a la entidad hija, ya que cada ocurrencia de ésta sólo puede aparecer una vez en la tabla.

Si la relación corresponde a una entidad débil con la entidad fuerte de la que depende, lo único que se debe hacer es añadir los atributos de la relación (si los tiene), a la tabla de la entidad débil, puesto que ésta ya contiene la clave ajena a la tabla de la entidad fuerte, que además de ayudarle a identificarse (formará parte de su clave primaria), expresa la relación.

Ejemplo 7.4 *Relación uno a muchos.*

El diagrama de la figura 7.4 contiene información de los profesores (código y nombre) y de los estudiantes (código y nombre). Algunos profesores tutorizan estudiantes y cada estudiante sólo puede ser tutorizado por un profesor.

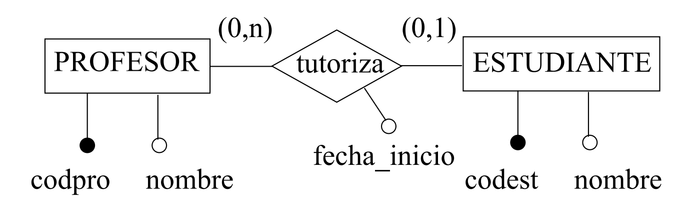


Figura 7.4: Relación de uno a muchos.

A continuación se muestran los dos posibles esquemas lógicos correspondientes a este diagrama:

(a) PROFESOR(codpro, nombre)
ESTUDIANTE(codest, nombre, codpro, fecha_inicio)
ESTUDIANTE.codpro es una clave ajena a PROFESOR, acepta nulos
Se debe cumplir la siguiente restricción:
(ESTUDIANTE.codpro IS NULL AND ESTUDIANTE.fecha_inicio IS NULL)
OR (ESTUDIANTE.codpro IS NOT NULL AND ESTUDIANTE.fecha_inicio IS NOT NULL)

(b) Otro modo de representar la relación es mediante una tabla aparte:

PROFESOR(codpro, nombre)
ESTUDIANTE(codest, nombre)
TUTORIZA(codest, codpro, fecha_inicio)
TUTORIZA.codest es una clave ajena a ESTUDIANTE, no acepta nulos
TUTORIZA.codpro es una clave ajena a PROFESOR, no acepta nulos

Relaciones binarias muchos a muchos

Para las relaciones binarias de muchos a muchos la única opción que existe es crear una tabla aparte para almacenar las ocurrencias de la relación. Esta tabla contendrá una clave ajena a cada una de las tablas correspondientes a las entidades participantes, además de incluir los atributos de la relación. Ninguna de las claves ajenas aceptará nulos. La clave primaria de esta tabla se determina en función de que la relación tenga o no atributos:

(a) Si la relación no tiene atributos, la clave primaria está formada por las dos claves ajenas (será una clave primaria compuesta).

(b) Si la relación tiene atributos, la clave primaria depende del significado de la relación. No hay que olvidar que las claves candidatas de una tabla

son restricciones que sus filas deben cumplir (sus valores no se pueden repetir) y, por lo tanto, será el significado de la relación (qué relaciones se pueden dar y cuáles no) el que nos ayudará a determinar las claves candidatas y, a partir de ellas, la clave primaria.

Ejemplo 7.5 *Relación muchos a muchos.*

El diagrama de la figura 7.5 contiene información de los médicos (código y nombre) y de los pacientes (código y nombre) de un centro médico, con información de las citas que éstos tienen concertadas. Se debe tener en cuenta que un paciente puede tener concertadas varias citas con el mismo médico.

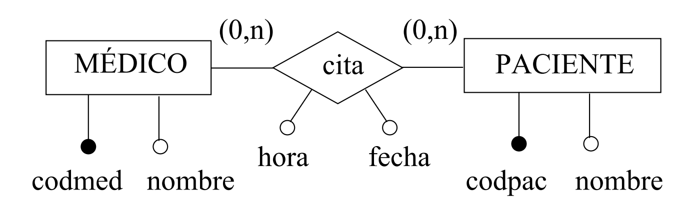


Figura 7.5: Relación de muchos a muchos.

A continuación, se muestra el esquema lógico correspondiente al diagrama anterior.

`MÉDICO(`<u>`codmed`</u>`, nombre)`
`PACIENTE(`<u>`codpac`</u>`, nombre)`
`CITA(codmed, codpac, fecha, hora)` ⇝ ¡falta escoger la clave primaria!
`CITA.codmed` es una clave ajena a `MÉDICO`, no acepta nulos
`CITA.codpac` es una clave ajena a `PACIENTE`, no acepta nulos

Para escoger la clave primaria de la tabla `CITA` se deben buscar antes las claves candidatas, que dependerán del significado de la relación:

- `(codmed, fecha, hora)` es una clave candidata, porque un médico no puede tener más de una cita el mismo día a la misma hora.

- `(codpac, fecha, hora)` es una clave candidata, porque un paciente no puede tener más de una cita el mismo día a la misma hora.

Nótese que `(codmed, codpac)` no es una clave candidata, ya que un mismo paciente puede tener varias citas con un mismo médico.

7.2.4. Jerarquías de generalización

En las jerarquías se denomina entidad madre, a la entidad genérica, y entidades hijas, a las subentidades. Hay tres opciones distintas para representar las jerarquías. La elección de la más adecuada se hará en función de su tipo (total o parcial, y exclusiva o superpuesta) y del tipo y frecuencia en los accesos a los datos. Estas opciones se presentan a continuación:

(a) Crear una tabla por cada entidad (madre e hijas). Las tablas de las entidades hijas heredan como clave primaria la clave primaria de la entidad madre. La clave primaria de las hijas es una clave ajena a la entidad madre. Esta representación se puede hacer para cualquier tipo de jerarquía, ya sea total o parcial, o exclusiva o superpuesta.

(b) Crear una tabla por cada entidad hija, heredando cada una los atributos de la entidad madre. Esta representación sólo puede hacerse para jerarquías totales y exclusivas.

(c) Integrar todas las entidades en una sola tabla, incluyendo en ella los atributos de la entidad madre, los atributos de todas las hijas y un atributo discriminativo para indicar el subconjunto al cual pertenece la entidad en consideración. Esta representación se puede utilizar para cualquier tipo de jerarquía. Si la jerarquía es superpuesta, el atributo discriminativo deberá ser multievaluado o bien se deberá incluir uno de estos atributos por cada subentidad.

Ejemplo 7.6 *Jerarquía de generalización.*

El diagrama de la figura 7.6 corresponde al ejemplo 6.7 del capítulo 6.

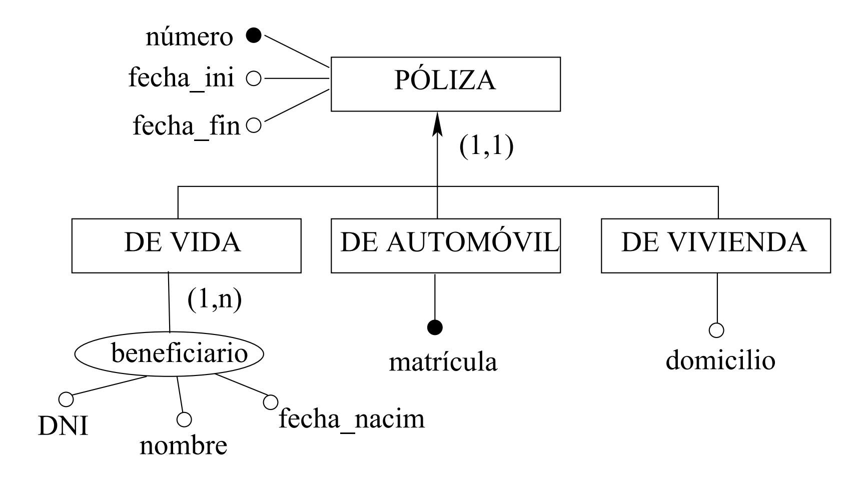


Figura 7.6: Ejemplo de jerarquía de generalización.

A continuación, se muestran los tres posibles esquemas lógicos correspondientes al diagrama anterior.

(a) `PÓLIZA(número, fecha_ini, fecha_fin)`
`PÓLIZA_VIDA(número, BENEFICIARIO(dni, nombre, fecha_nacim))`
`PÓLIZA_VIDA.número` es una clave ajena a `PÓLIZA`
`PÓLIZA_AUTOMÓVIL(número, matrícula)`
`PÓLIZA_AUTOMÓVIL.matrícula` es una clave alternativa
`PÓLIZA_AUTOMÓVIL.número` es una clave ajena a `PÓLIZA`
`PÓLIZA_VIVIENDA(número, domicilio)`
`PÓLIZA_VIVIENDA.número` es una clave ajena a `PÓLIZA`

(b) `PÓLIZA_VIDA(número, fecha_ini, fecha_fin, BENEFICIARIO(dni, nombre, fecha_nacim))`
`PÓLIZA_AUTOMÓVIL(número, fecha_ini, fecha_fin, matrícula)`
`PÓLIZA_AUTOMÓVIL.matrícula` es una clave alternativa
`PÓLIZA_VIVIENDA(número, fecha_ini, fecha_fin, domicilio)`

(c) `PÓLIZA(número, fecha_ini, fecha_fin, tipo, BENEFICIARIO(dni, nombre, fecha_nacim), matrícula, domicilio)`
`PÓLIZA.tipo` $\in$ `{'vida','automóvil','vivienda'}` $\rightsquigarrow$ atributo discriminativo
`PÓLIZA.matrícula` es una clave alternativa
`PÓLIZA.matrícula, PÓLIZA.domicilio` aceptan nulos

Una vez obtenidas las tablas con sus atributos, claves primarias, claves alternativas y claves ajenas, deben normalizarse. La normalización se utiliza para mejorar el esquema lógico, de modo que satisfaga ciertas restricciones que eviten la duplicidad de datos. La normalización garantiza que el esquema resultante se encuentra más próximo al modelo de la empresa, que es consistente y que tiene la mínima redundancia y la máxima estabilidad.

7.2.5. Normalización

La normalización es una técnica para diseñar la estructura lógica de los datos de un sistema de información en el modelo relacional, desarrollada por E. F. Codd en 1972. Es una estrategia de diseño de abajo a arriba: se parte de los atributos y éstos se van agrupando en tablas según su afinidad. Aquí no se utilizará la normalización como una técnica de diseño de bases de datos, sino como una etapa posterior a la correspondencia entre el esquema conceptual y el esquema lógico, que elimine las dependencias entre atributos no deseadas.

En la mayoría de las ocasiones, una base de datos completamente normalizada no proporciona la máxima eficiencia; sin embargo, el objetivo en esta etapa es conseguir una base de datos normalizada por las siguientes razones:

- Un esquema normalizado organiza los datos de acuerdo a sus dependencias funcionales, es decir, de acuerdo a sus relaciones lógicas.

- El esquema lógico no tiene por qué ser el esquema final. Debe representar lo que el diseñador entiende sobre la naturaleza y el significado de los datos de la empresa. Si se establecen unos objetivos en cuanto a prestaciones, el diseño físico cambiará el esquema lógico de modo adecuado. Una posibilidad es que algunas tablas normalizadas se desnormalicen. Pero la desnormalización no implica que se haya malgastado tiempo normalizando, ya que mediante este proceso el diseñador aprende más sobre el significado de los datos. De hecho, la normalización obliga a entender completamente cada uno de los atributos que se han de representar en la base de datos.

- Un esquema normalizado es robusto y carece de redundancias, por lo que está libre de ciertas anomalías que las redundancias pueden provocar cuando se actualiza la base de datos.

- Los equipos informáticos de hoy en día son cada vez más potentes, por lo que en ocasiones es más razonable implementar bases de datos fáciles de manejar (las normalizadas), a costa de un tiempo adicional de proceso.

- La normalización produce bases de datos con esquemas flexibles que pueden extenderse con facilidad.

De lo que se trata es de obtener un conjunto de tablas que se encuentren en la forma normal de Boyce-Codd. Para ello, hay que pasar por la primera, segunda y tercera formas normales.

Dependencia funcional

Uno de los conceptos fundamentales en la normalización es el de *dependencia funcional*. Una dependencia funcional es una relación entre atributos de una misma tabla. Si x e y son atributos de la relación R, se dice que y es funcionalmente dependiente de x (se denota por $x \longrightarrow y$) si cada valor de x tiene asociado un solo valor de y (x e y pueden constar de uno o varios atributos). A x se le denomina *determinante*, ya que x determina el valor de y. Se dice que el atributo y es *completamente dependiente* de x si depende funcionalmente de x y no depende de ningún subconjunto de x.

La dependencia funcional es una noción semántica. Si hay o no dependencias funcionales entre atributos, no lo determina una serie abstracta de reglas, sino, más bien, los modelos mentales del usuario y las reglas de negocio de la organización o empresa para la que se desarrolla el sistema de información. Cada dependencia funcional es una restricción y representa una relación de uno a muchos (o de uno a uno).

En el proceso de normalización debe irse comprobando que cada tabla cumple una serie de reglas que se basan en la clave primaria y las dependencias funcionales. Cada regla que se cumple aumenta el grado de normalización. Si

una regla no se cumple, la tabla se debe descomponer en varias tablas que sí la cumplan.

La normalización se lleva a cabo en una serie de pasos. Cada paso corresponde a una forma normal que tiene unas propiedades. Conforme se va avanzando en la normalización, las tablas tienen un formato más estricto (más fuerte) y, por lo tanto, son menos vulnerables a las anomalías de actualización. El modelo relacional sólo requiere un conjunto de tablas en primera forma normal (en caso contrario no se pueden implementar). Las restantes formas normales son opcionales. Sin embargo, para evitar las anomalías de actualización, es recomendable llegar al menos a la tercera forma normal.

Primera forma normal

Una tabla está en primera forma normal (1FN) si, y sólo si, todos los dominios de sus atributos contienen valores atómicos, es decir, no hay *grupos repetitivos*. Un grupo repetitivo es un atributo que puede tener múltiples valores para cada fila de la relación. Son los atributos que tienen forma de tabla.

Si una tabla no está en 1FN, hay que eliminar de ella los grupos repetitivos. La forma de eliminar los grupos repetitivos consiste en poner cada uno de ellos como una tabla aparte, heredando la clave primaria de la tabla en la que se encontraban. La clave primaria de esta nueva tabla estará formada por la combinación de la clave primaria que tenía cuando era un grupo repetitivo y la clave primaria que ha heredado en forma de clave ajena. Se dice que conjunto de tablas se encuentra en 1FN si ninguna de ellas tiene grupos repetitivos.

Ejemplo 7.7 *Pasar una tabla a 1FN.*

La tabla `PRODUCTO` que se muestra a continuación no se encuentra en 1FN, ya que tiene un grupo repetitivo:

```
PRODUCTO(codprod, nombre, VERSIÓN(número, fecha, ventas))
```

Para pasarla a 1FN se debe eliminar el grupo repetitivo:

```
PRODUCTO(codprod, nombre)
VERSIÓN(codprod, número, fecha, ventas)
VERSIÓN.codprod es una clave ajena a PRODUCTO
```

Segunda forma normal

Una tabla está en segunda forma normal (2FN) si, y sólo si, está en 1FN y, además, cada atributo que no forma parte de la clave primaria es completamente dependiente de la clave primaria.

La 2FN se aplica a las tablas que tienen claves primarias compuestas por dos o más atributos. Si una tabla está en 1FN y su clave primaria es simple (tiene un solo atributo), entonces también está en 2FN. Las tablas que no están en 2FN pueden sufrir anomalías cuando se realizan actualizaciones sobre ellas.

Para pasar una tabla en 1FN a 2FN hay que eliminar las dependencias parciales de la clave primaria. Para ello, se eliminan los atributos que son funcionalmente dependientes y se ponen en una nueva tabla con una copia de su determinante. Su determinante estará formado por los atributos de la clave primaria de los que depende.

Ejemplo 7.8 *Pasar una tabla en 1FN a 2FN.*

En la tabla `INSCRIPCIÓN` que aparece a continuación existe una dependencia funcional parcial de la clave primaria:

`INSCRIPCIÓN(estudiante, actividad, precio)`
Dependencia funcional parcial: `actividad` $\longrightarrow$ `precio`

Esta dependencia existe porque cada actividad tiene un precio, independientemente del estudiante que se inscriba. Las anomalías que se pueden producir si se mantiene esta dependencia dentro de la tabla son varias. Por una parte, no es posible conocer el precio de una actividad si no hay personas inscritas, ya sea porque no se ha inscrito ninguna o porque todas las que lo están cancelan su inscripción. Por otra parte, y que es aún más grave, si se cambia el precio de una actividad y no se cambia para todas las personas inscritas, se tendrá una falta de integridad ya que habrá dos precios para la misma actividad, uno correcto y otro erróneo. Para pasar la tabla a 2FN se debe eliminar el atributo de la dependencia parcial, que se lleva una copia de su determinante:

`ACTIVIDAD(actividad, precio)`
`INSCRIPCIÓN(estudiante, actividad)`
`INSCRIPCIÓN.actividad` es una clave ajena a `ACTIVIDAD`

De este modo se evitan las anomalías citadas anteriormente: puede conocerse el precio de las actividades sin haber inscripciones y, puesto que el precio sólo está almacenado una vez, si se cambia éste, será el mismo para todas las inscripciones.

Tercera forma normal

Una tabla está en tercera forma normal (3FN) si, y sólo si, está en 2FN y, además, cada atributo que no forma parte de la clave primaria no depende transitivamente de la clave primaria. La dependencia $x \longrightarrow z$ es transitiva si existen las dependencias $x \longrightarrow y$, $y \longrightarrow z$, siendo x, y, z atributos o conjuntos de atributos de una misma tabla.

Aunque las relaciones en 2FN tienen menos redundancias que las relaciones en 1FN, todavía pueden sufrir anomalías frente a las actualizaciones. Para pasar una relación en 2FN a 3FN hay que eliminar las dependencias transitivas. Para ello, se eliminan los atributos que dependen transitivamente y se ponen en una nueva relación con una copia de su determinante (el atributo o atributos no clave de los que depende).

Ejemplo 7.9 *Pasar una tabla en 2*FN *a 3*FN*.*

En la tabla `HABITA` existe una dependencia funcional transitiva:

> `HABITA(inquilino, edificio, alquiler)`
> Dependencia funcional transitiva: `edificio` $\longrightarrow$ `alquiler`

Esta dependencia existe porque cada edificio tiene un alquiler, independientemente del inquilino que lo habite. Una vez más, mantener esta dependencia dentro de la tabla puede dar lugar a diversas anomalías: no es posible conocer el alquiler de un edificio si no hay inquilinos y si se modifica el precio del alquiler de un edificio sólo para algunos inquilinos se viola una regla del negocio, ya que todos los inquilinos del mismo edificio deben pagar lo mismo. Para pasar la tabla a 3FN se debe eliminar el atributo de la dependencia transitiva, que se lleva una copia de su determinante:

> `ALQUILER(edificio, alquiler)`
> `HABITA(inquilino, edificio)`
> `HABITA.edificio` es una clave ajena a `ALQUILER`

Descomponiendo la tabla de este modo, se evitan las anomalías que se han citado.

Forma normal de Boyce-Codd

Una tabla está en la forma normal de Boyce-Codd (BCFN) si, y sólo si, todo determinante es una clave candidata.

La 2FN y la 3FN eliminan las dependencias parciales y las dependencias transitivas de la clave primaria. Pero este tipo de dependencias todavía pueden existir sobre otras claves candidatas, si éstas existen. La BCFN es más fuerte que la 3FN, por lo tanto, toda tabla en BCFN está en 3FN.

La violación de la BCFN es poco frecuente ya que se da bajo ciertas condiciones que raramente se presentan. Se debe comprobar si una tabla viola la BCFN en caso de tener dos o más claves candidatas compuestas que tienen al menos un atributo en común.

Cómo saber si se ha hecho bien la normalización

En primer lugar, hay que fijarse en que las dependencias funcionales no deseadas han desaparecido. Las únicas dependencias que deben quedar son las que son de la clave primaria completa.

Al normalizar una tabla (2FN y 3FN) lo que se hace es obtener distintas proyecciones de ella, para repartir sus columnas en varias tablas de modo que se eliminen las dependencias no deseadas (no son más que redundancias de datos). Por lo tanto, el conjunto de tablas que se obtiene al normalizar debe permitir recuperar la tabla original haciendo concatenaciones (JOIN). Si nos fijamos en el ejemplo 7.8, las proyecciones que se han hecho son:

```
ACTIVIDAD := SELECT actividad,precio FROM INSCRIPCIÓN;
INSCRIPCIÓN := SELECT estudiante,actividad FROM INSCRIPCIÓN;
```

Y a partir de ellas es posible recuperar la tabla original:

```
SELECT * FROM ACTIVIDAD JOIN INSCRIPCIÓN USING(actividad);
```

Algo que puede ser también de utilidad, para comprobar si se ha normalizado correctamente, es que la clave primaria de cada tabla debe ser distinta. Por ejemplo, supongamos que la tabla original de inscripciones tenía 3500 filas, que corresponden a las inscripciones de 2800 estudiantes en 32 actividades distintas. La nueva tabla de inscripciones tendrá 3500 filas, mientras que la nueva tabla de actividades tendrá 32 filas. La tabla de inscripciones mantiene su clave primaria y, por lo tanto, mantiene su número de filas. La tabla de actividades tiene como clave primaria la columna de la que salía una dependencia no deseada en la tabla original, tendrá tantas filas como actividades distintas existen.

Si no se hacen bien las proyecciones y se obtiene más de una tabla con la misma clave primaria, las flechas no deseadas seguirán estando presentes, quizás en otra tabla, continuando presentes las dependencias funcionales no deseadas.

7.3. Restricciones de integridad

La definición de las restricciones de integridad se lleva a cabo en la etapa del diseño lógico. Las restricciones son reglas que se quiere imponer para proteger la base de datos, de modo que no pueda llegar a un estado inconsistente en el que los datos no reflejen la realidad o sean contradictorios. Hay cinco tipos de restricciones de integridad.

(a) *Datos requeridos.* Algunos atributos deben contener valores en todo momento, es decir, no admiten nulos.

(b) *Restricciones de dominios.* Todos los atributos tienen un dominio asociado, que es el conjunto de valores que cada atributo puede tomar.

(c) *Integridad de entidades.* El identificador de una entidad no puede ser nulo, por lo tanto, las claves primarias de las tablas no admiten nulos.

(d) *Integridad referencial.* Una clave ajena enlaza cada fila de la tabla hija con la fila de la tabla madre que tiene el mismo valor en su clave primaria. La integridad referencial dice que si una clave ajena tiene valor (si es no nula), ese valor debe ser uno de los valores de la clave primaria a la que referencia. Hay varios aspectos a tener en cuenta sobre las claves ajenas para lograr que se cumpla la integridad referencial:

1. *¿Admite nulos la clave ajena?* Cada clave ajena expresa una relación. Si la participación de la entidad hija en la relación es obligatoria (cardinalidad mínima 1), entonces la clave ajena no admite nulos; si es opcional (cardinalidad mínima 0), la clave ajena debe aceptar nulos.
2. *¿Qué hacer cuando se quiere borrar una ocurrencia de la entidad madre que tiene alguna hija?* Esto es lo mismo que preguntarse qué hacer cuando se quiere borrar una fila que está siendo referenciada por otra fila a través de una clave ajena. Hay varias respuestas posibles:
   - *Restringir:* no se pueden borrar filas que están siendo referenciadas por otras filas.
   - *Propagar:* se borra la fila deseada y se propaga el borrado a todas las filas que le hacen referencia.
   - *Anular:* se borra la fila deseada y todas las referencias que tenía se ponen, automáticamente, a nulo (esta opción sólo es válida si la clave ajena acepta nulos).
   - *Valor por defecto:* se borra la fila deseada y todas las referencias toman, automáticamente, el valor por defecto (esta opción sólo es válida si se ha especificado un valor por defecto para la clave ajena).
3. *¿Qué hacer cuando se quiere modificar la clave primaria de una fila que está siendo referenciada por otra fila a través de una clave ajena?* Las respuestas posibles son las mismas que en el caso anterior. Cuando se escoge propagar, se actualiza la clave primaria en la fila deseada y se propaga el cambio a los valores de clave ajena que le hacían referencia.

(e) *Restricciones y reglas de negocio.* Cualquier operación que se realice sobre los datos debe cumplir las restricciones y las reglas que impone el funcionamiento de la empresa. Hablamos de restricciones cuando se dan ciertas condiciones que no deben violarse y hablamos de reglas de negocio cuando se requiere la ejecución automática de ciertas acciones ante determinados eventos.

Todas las restricciones de integridad establecidas en este paso se deben reflejar en la documentación del esquema lógico para que puedan ser tenidas en cuenta durante la fase del diseño físico.

7.4. Desnormalización

Una de las tareas que se realizan en el diseño lógico, después de obtener un esquema lógico normalizado, es la de considerar la introducción de redundancias controladas y otros cambios en el esquema. En ocasiones puede ser conveniente relajar las reglas de normalización introduciendo redundancias de forma controlada con objeto de mejorar las prestaciones del sistema.

En la etapa del diseño lógico se recomienda llegar, al menos, hasta la tercera forma normal para obtener un esquema con una estructura consistente y sin redundancias. Pero a menudo sucede que las bases de datos así normalizadas no proporcionan la máxima eficiencia, con lo que es necesario volver atrás y desnormalizar algunas tablas, sacrificando los beneficios de la normalización para mejorar las prestaciones. Es importante hacer notar que la desnormalización sólo debe realizarse cuando se estime que el sistema no puede alcanzar las prestaciones deseadas. Y desde luego, el que en ocasiones sea necesario desnormalizar no implica eliminar la fase de normalización del diseño lógico ya que la normalización obliga al diseñador a entender completamente cada uno de los atributos que se han de representar en la base de datos.

Además hay que tener en cuenta los siguientes factores:

- La desnormalización hace que la implementación sea más compleja.
- La desnormalización hace que se sacrifique la flexibilidad.
- La desnormalización puede hacer que los accesos a datos sean más rápidos, pero ralentiza las actualizaciones.

Por regla general, la desnormalización puede ser una opción viable cuando las prestaciones que se obtienen no son las deseadas y las tablas involucradas se actualizan con poca frecuencia pero se consultan muy a menudo. Las redundancias que se pueden incluir al desnormalizar son de varios tipos: se pueden introducir datos derivados (calculados a partir de otros datos), se pueden duplicar atributos o se puede hacer concatenaciones (`JOIN`) de tablas. El incluir redundancias dependerá del coste adicional de almacenarlas y mantenerlas consistentes, frente al beneficio que se consigue al realizar consultas.

No se puede establecer una serie de reglas que determinen cuándo desnormalizar tablas, pero hay algunas situaciones bastante comunes en donde puede considerarse esta posibilidad:

- *Combinar relaciones de uno a uno.* Esto puede ser conveniente cuando hay tablas involucradas en relaciones de uno a uno, se accede a ellas de manera conjunta con frecuencia y casi no se accede a ellas por separado.

- *Tablas de referencia.* Las tablas de referencia (*lookup tables*) son listas de valores posibles de una o varias columnas de la base de datos. La lista normalmente consta de una descripción (valor) y un código. Este tipo de tablas son un caso de relación de uno a muchos y con ellas es muy fácil validar los datos. Mediante ellas se puede ahorrar espacio en las tablas donde se usan los valores de referencia ya que se puede escribir sólo el código (como una clave ajena) y no el valor en sí (descripción).

  Si las tablas de referencia se utilizan a menudo en las consultas, se puede considerar la introducción de la descripción junto con el código en la tabla hijo, manteniendo la tabla de referencia para validación de datos cuando éstos se introducen en la base de datos. De esta forma se evitan los `JOIN` con la tabla de referencia al hacer las consultas. En este caso, se puede eliminar la restricción de la clave ajena ya que no es necesario mantener la integridad referencial, al copiarse los valores en la tabla hijo.

- *Duplicar atributos no clave en relaciones de uno a muchos.* Para evitar operaciones de `JOIN` entre tablas, se pueden incluir atributos de la tabla madre en la tabla hija de las relaciones de uno a muchos.

- *Duplicar claves ajenas en relaciones de uno a muchos.* Para evitar operaciones de `JOIN`, se pueden incluir claves ajenas de una tabla en otra tabla con la que se relaciona (habrá que tener en cuenta ciertas restricciones).

- *Duplicar atributos en relaciones de muchos a muchos.* Durante el diseño lógico se crea una nueva tabla para almacenar las ocurrencias de una relación de muchos a muchos, de modo que si se quiere obtener la información de la relación de muchos a muchos, se tiene que realizar el `JOIN` de tres tablas. Para evitar algunos de estos `JOIN` puede incluirse algunos de los atributos de las tablas originales en la tabla intermedia.

- *Introducir grupos repetitivos.* Los grupos repetitivos se eliminan en el primer paso de la normalización para conseguir la primera forma normal. Estos grupos se eliminan introduciendo una nueva tabla, generando una relación de uno a muchos. A veces, puede ser conveniente reintroducir los grupos repetitivos para mejorar las prestaciones. El grupo repetitivo debe desplegarse dentro de la tabla, por lo que la clave primaria de la tabla original deberá incluir a la clave primaria del grupo repetitivo.

- *Partir tablas.* Las tablas se pueden partir horizontalmente (por casos) o verticalmente (por atributos) de modo que a partir de una tabla grande, que tiene datos a los que no se accede con frecuencia, se obtengan tablas más pequeñas, algunas de las cuales contienen sólo datos a los que sí se accede muy a menudo.

Todas las transformaciones y redundancias que se introduzcan en este paso se deben documentar y razonar. El esquema lógico se debe actualizar para reflejar los cambios introducidos.

7.5. Reglas de comportamiento de las claves ajenas

Para cada clave ajena que aparece en el esquema lógico se deben especificar sus reglas de comportamiento ante el borrado y la modificación de la clave primaria a la que referencia. Además, para cada una, se debe establecer si acepta nulos o no. En gran medida, las reglas de las claves ajenas son establecidas por los propietarios de los datos.

Las claves ajenas y sus reglas se han estudiado en el capítulo 2 (apartado 2.4.3), por lo que en este apartado se estudiará el establecimiento de las mismas con un caso práctico:

```
EMPLEADO(codemp, nombre, matrícula, fecha_ini)
VEHÍCULO(matrícula, modelo)
```

En esta base de datos hay datos de empleados y de vehículos. Cada empleado conduce un solo vehículo y cada vehículo puede ser conducido por distintos empleados. En este caso se ha incluido la clave ajena que representa la relación, en la tabla que contiene la información de los empleados, de modo que cada empleado hace referencia al vehículo que conduce. A continuación, se plantean las preguntas que hay que responder para establecer las reglas de las claves ajenas:

- *¿Acepta nulos la clave ajena?*, es decir, *¿puede haber algún empleado que no conduzca ningún vehículo?* La respuesta aparece en el esquema conceptual, en la cardinalidad mínima con la que participa la entidad `EMPLEADO` en la relación: si es 0 significa que sí puede haber empleados sin vehículo, por lo que la clave ajena debe aceptar nulos; si es 1 significa que todo empleado debe conducir algún vehículo, por lo que en este caso no debe aceptar nulos.

- *¿Cuál es la regla de borrado?*, es decir, *¿qué hacer cuando se intenta borrar un vehículo que es conducido por algún empleado?* Las posibles respuestas son:

  - *Propagar*: se borra el vehículo (se elimina su fila de la tabla) y se borran los empleados que lo conducen (también se borran las filas que hacen referencia a ese vehículo).
  - *Restringir*: no se puede borrar un vehículo que es conducido por algún empleado. En este caso, lo recomendable es pedir que el propietario de los datos especifique un procedimiento a seguir: dar sólo un aviso al usuario; dar un aviso y mostrar los datos del empleado dando la posibilidad de cambiarle el vehículo; etc.
  - *Anular*: se borra el vehículo y en las filas de los empleados que lo conducen, la clave ajena, que contenía la matrícula del vehículo,

se pone a nulo. Esta opción sólo es posible cuando la clave ajena acepta nulos.

- *Valor por defecto*: se borra el vehículo y en las filas de los empleados que lo conducen, la clave ajena, que contenía la matrícula del vehículo, se pone el valor por defecto. Esta opción sólo es posible cuando la clave ajena tiene un valor por defecto.

- *¿Cuál es la regla de modificación?*, es decir, *¿qué hay que hacer cuando se intenta modificar la matrícula de un vehículo que es conducido por algún empleado?* Las posibles respuestas son:

  - *Propagar*: se modifica la matrícula del vehículo y en las filas de los empleados que lo conducen se cambia el valor de la clave ajena (la matrícula) para que le siga haciendo referencia. Al fin y al cabo, el vehículo es el mismo, sólo ha cambiado el valor de una de sus propiedades (quizá porque se tecleó mal al introducirla).
  - *Restringir*: no se puede modificar la matrícula de un vehículo que es conducido por algún empleado. En este caso lo normal es pedir que el propietario de los datos especifique un procedimiento a seguir: dar un aviso al usuario, etc.
  - *Anular*: se modifica la matrícula del vehículo y en las filas de los empleados que lo conducen, la clave ajena, que contenía la matrícula del vehículo, se pone a nulo. De nuevo, esta opción sólo es posible cuando la clave ajena acepta nulos.
  - *Valor por defecto*: se modifica la matrícula del vehículo y en las filas de los empleados que lo conducen, la clave ajena, que contenía la matrícula del vehículo, toma el valor por defecto. De nuevo, esta opción sólo es posible cuando la clave ajena tiene un valor por defecto.

  Algunos SGBD no permiten especificar la regla de modificación. Esto es así porque si se necesita, es fácil implementarla mediante disparadores, pero no se ha estimado necesario ya que una clave primaria bien elegida será una clave que nunca cambiará de valor. Las claves primarias no deben ser columnas que representen propiedades de las entidades, sino columnas sin significado, que se pueden añadir a propósito, para las que van generándose valores únicos de manera automática, cuya función es solamente la de identificar las filas. Ya que este tipo de claves primarias se generan de modo automático, nunca cambian de valor (ni siquiera el usuario necesita saber que existen) por lo que este tipo de operación (modificación) no suele realizarse nunca.

Como se ha visto, las respuestas a las cuestiones anteriores están en los usuarios de los datos. Sin embargo, hay ocasiones en las que es el diseñador

quien debe decidir las reglas de determinadas claves ajenas. Vemos aquí cuáles son esas ocasiones:

- *Jerarquías.* Cuando se escoge representar una jerarquía del modo más general (el que funciona para todo tipo de jerarquía), se introduce una tabla por la entidad genérica y una tabla por cada subentidad. Las tablas correspondientes a las subentidades tienen, cada una, una clave ajena a la tabla correspondiente a la entidad. Esta clave ajena será también una clave candidata (podrá ser la clave primaria o podrá serlo cualquier otro identificador alternativo). Pues bien, esta clave ajena que tiene cada tabla de subentidad no acepta nulos y la regla del borrado será, por lo general, propagar. Esto debe ser así porque para el propietario de la información, la jerarquía del esquema conceptual es tan sólo una clasificación: si quiere borrar una ocurrencia de una entidad, no se le puede decir que no puede hacerlo (restringir) por el hecho de que esa ocurrencia haya sido clasificada de algún modo.

- *Atributos con múltiples valores.* Cuando una entidad tiene un atributo que puede tener varios valores (cuando la cardinalidad máxima del atributo es n), tras la normalización, se tiene una tabla que contendrá los distintos valores del atributo para cada ocurrencia de la entidad. Por ejemplo, podemos tener una entidad `EMPLEADO` con un atributo con múltiples valores en donde se indiquen los títulos académicos que tiene cada empleado. Este atributo dará lugar a una tabla que tendrá una clave ajena a la tabla de empleados; esta clave ajena formará parte de la clave primaria junto con el nombre del título, por ejemplo. Esta clave ajena no aceptará nulos y la regla de borrado será siempre propagar. En realidad, esta tabla ha aparecido porque en el modelo relacional es así como se representan los atributos con múltiples valores, mediante una nueva tabla. Para el usuario, el empleado es una entidad, pero en la base de datos la información se ha repartido en varias tablas. En este caso, no tiene sentido restringir el borrado. Lo que debe hacerse es que cuando un usuario intenta borrar una ocurrencia de una entidad, debe propagarse el borrado de ésta a cualquier otra tabla que almacene propiedades del empleado que hayan surgido a causa de atributos con múltiples valores.

7.6. Cuestiones adicionales

Una vez obtenido el esquema lógico, éste debe validarse frente a las transacciones de los usuarios. El objetivo de este paso es validar el esquema lógico para garantizar que puede soportar las transacciones requeridas por los correspondientes usuarios. Estas transacciones se encontrarán en las especificaciones de requisitos de usuario. Lo que debe hacerse es tratar de realizar las transacciones de forma manual utilizando el diagrama entidad-relación, el diccionario de datos y las conexiones que establecen las claves ajenas de las tablas. Si todas las transacciones se pueden realizar, el esquema queda validado. Pero si alguna transacción no se puede realizar, seguramente será porque alguna entidad, relación o atributo no se ha incluido en el esquema.

Además, para garantizar que cada esquema lógico local es una fiel representación de la vista del usuario lo que se debe hacer es comprobar con él que lo reflejado en el esquema y en la documentación es correcto y está completo.

El esquema lógico refleja la estructura de los datos a almacenar que maneja la empresa. Un *diagrama de flujo de datos* muestra cómo se mueven los datos entre los procesos y los almacenes en donde se guardan. Si se han utilizado diagramas de flujo de datos para modelar las especificaciones de requisitos de usuario, se pueden utilizar para comprobar la consistencia y completitud del esquema lógico desarrollado. Para ello:

- Cada almacén de datos debe corresponder con una o varias entidades completas.
- Los atributos en los flujos de datos deben corresponder a alguna entidad.

Una última cuestión a tener en cuenta es la de estudiar el crecimiento futuro. En este paso, se trata de comprobar que el esquema obtenido puede acomodar los futuros cambios en los requisitos con un impacto mínimo. Si el esquema lógico se puede extender fácilmente, cualquiera de los cambios previstos se podrá incorporar al mismo con un efecto mínimo sobre los usuarios existentes.

7.7. Ejemplos

En este apartado se obtendrá el esquema lógico correspondiente a los dos ejemplos presentados en el apartado 6.3 del capítulo 6 de diseño conceptual.

Ejemplo 7.10 *Asociación de cines.*

Comenzamos la obtención de las tablas a partir de las entidades:

```
CINES(nombre, calle, número, teléfono, TARIFA(tipo, precio))
PELÍCULAS(título, director, protagonista1, protagonista2,
```

protagonista3, género, clasificación)
Los atributos PELÍCULAS.protagonista2 y PELÍCULAS.protagonista3 aceptan nulos.

Puesto que se debe almacenar el nombre de hasta tres protagonistas, se ha escogido representar el atributo como tres columnas, en lugar de hacerlo como un atributo multievaluado. De esta forma, toda la información de una película está en una misma fila. Nótese que los nombres de las entidades se han puesto en plural al obtener las tablas correspondientes.[2]

Veamos ahora cómo se debe representar la relación entre los cines y las películas que pasan (la cartelera):

CARTELERA(nombre_cine, título_película, PASES(hora))
CARTELERA.nombre_cine es clave ajena a CINES
CARTELERA.título_película es clave ajena a PELÍCULAS

Se han renombrado las columnas que son claves ajenas, de modo que llevan detrás el nombre de la tabla a la que hacen referencia.

Una vez obtenidas las tablas, se debe pasar a la normalización. Las tablas CINES y CARTELERA no están en 1FN, por lo que debemos normalizarlas:

CINES(nombre, calle, número, teléfono)
TARIFA(nombre_cine, tipo, precio)
TARIFA.nombre_cine es clave ajena a CINES

CARTELERA(nombre_cine, título_película)
CARTELERA.nombre_cine es clave ajena a CINES
CARTELERA.título_película es clave ajena a PELÍCULAS
PASES(nombre_cine, título_película, hora)
(PASES.nombre_cine, PASES.título_película) es clave ajena a CARTELERA

Nótese que la clave ajena de PASES a CARTELERA es una clave ajena compuesta.

Las tablas obtenidas están en 1FN y también en 2FN y 3FN, al no haber dependencias funcionales no deseadas, por lo que el esquema lógico contiene ya las tablas normalizadas. El diagrama de la figura 7.7 muestra las tablas de la base de datos. El recuadro superior de cada tabla contiene la clave primaria. Mediante flechas se han indicado las claves ajenas y sobre estas flechas, se han indicado las reglas de comportamiento de las mismas.

[2] Esta es una cuestión de notación. El diseñador debe escoger una notación para nombrar tablas, columnas y claves.

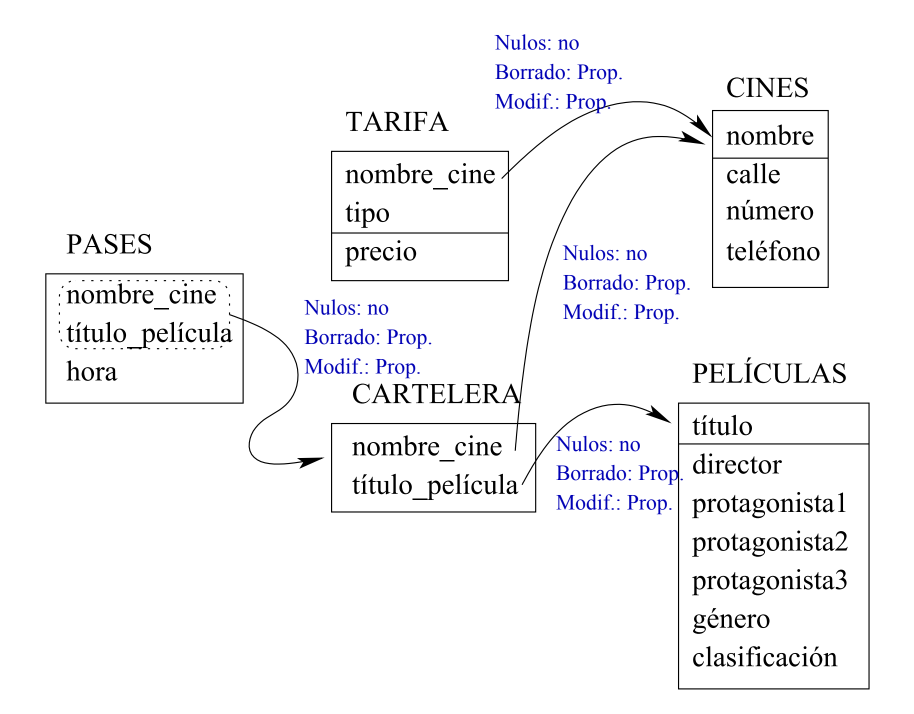


Figura 7.7: Esquema relacional correspondiente al caso de la asociación de cines.

Ejemplo 7.11 *Catálogo de un portal web.*

Comenzamos la obtención de las tablas a partir de las entidades:

```
TEMAS(tema, PALABRAS(palabra, importancia),
CONSULTAS(ip, fecha_hora))
PÁGINAS(url, título)
VOLUNTARIOS(email, nombre)
```

Por comodidad, pasamos ahora la tabla `TEMAS` a 1FN, ya que debemos incluir columnas en ella para representar las relaciones.

`TEMAS(tema)`
`PALABRAS(tema, palabra, importancia)`
`PALABRAS.tema` es clave ajena a `TEMAS`
`CONSULTAS(tema, ip, fecha_hora)`
`CONSULTAS.tema` es clave ajena a `TEMAS`

Incluimos ahora las relaciones en el esquema lógico:

`TEMAS(tema, tema_padre)`
`TEMAS.tema_padre` es clave ajena a `TEMAS`
`CONTENIDO(url_página, tema, prioridad)`
`(CONTENIDO.url_página, CONTENIDO.prioridad)` es clave alternativa
`CONTENIDO.url_página` es clave ajena a `PÁGINAS`

```
CONTENIDO.tema es clave ajena a TEMAS
EVALUACIONES(email_voluntario, url_página, fecha, calificación)
EVALUACIONES.email_voluntario es clave ajena a VOLUNTARIOS
EVALUACIONES.url_página es clave ajena a PÁGINAS
```

La clave primaria de la tabla EVALUACIONES no permite que haya más de una evaluación de un mismo voluntario con una misma página. Se deberá establecer un mecanismo que, ante una nueva evaluación de una página ya evaluada antes por el mismo voluntario, sustituya la evaluación previa por la que se acabe de realizar.

Las tablas que se han obtenido están en 3FN (no hay dependencias funcionales no deseadas). La figura 7.8 muestra las tablas que se acaban de obtener, con las claves primarias y las claves ajenas.

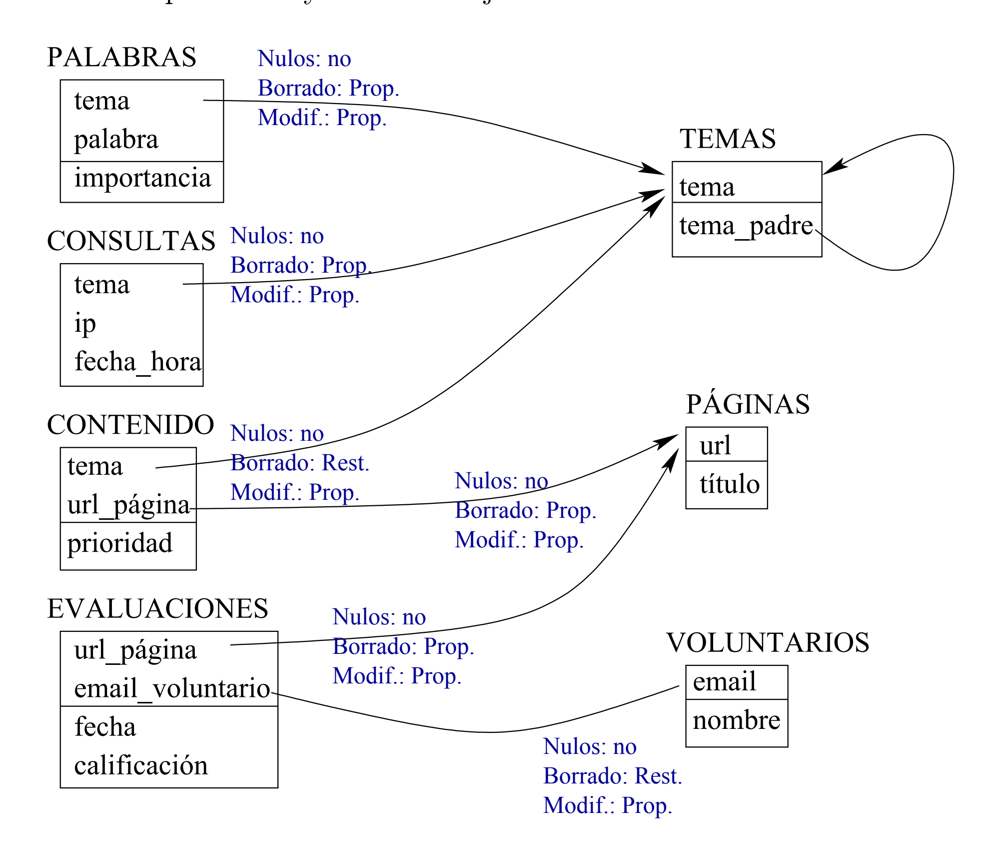


Figura 7.8: Esquema relacional correspondiente al caso del catálogo web.

Capítulo 8

Diseño físico en SQL

Introducción y objetivos

El diseño físico es el proceso de producir la descripción de la implementación de la base de datos en memoria secundaria, a partir del esquema lógico obtenido en la etapa anterior. Para especificar dicha implementación se debe determinar las estructuras de almacenamiento y escoger los mecanismos necesarios para garantizar un acceso eficiente a los datos. Puesto que el esquema lógico utiliza el modelo relacional, la implementación del diseño físico se realizará en SQL.

Al finalizar este capítulo, el estudiante debe ser capaz de:

- Traducir el esquema lógico de una base de datos dada a un conjunto de sentencias SQL de creación de tablas que la implementen fielmente.
- Acudir a los manuales del SGBD escogido para la implementación y obtener en ellos toda la información necesaria para llevar a cabo la implementación sobre el mismo (sintaxis del lenguaje, los tipos de datos, etc.).
- Escoger las organizaciones de ficheros más apropiadas en función de las que tenga disponible el SGBD que se vaya a utilizar.
- Decidir qué índices deben crearse con el objetivo de aumentar las prestaciones en el acceso a los datos.
- Diseñar las vistas necesarias para proporcionar seguridad y facilitar el manejo de la base de datos.

8.1. Metodología de diseño

Mientras que en el diseño lógico se especifica qué se guarda, en el diseño físico se especifica cómo se guarda.

Para llevar a cabo esta etapa se debe haber decidido cuál es el SGBD que se va a utilizar, ya que el esquema físico se adapta a él. El diseñador debe conocer muy bien toda la funcionalidad del SGBD concreto y también el sistema informático sobre el que éste va a trabajar.

El diseño físico no es una etapa aislada, ya que algunas decisiones que se tomen durante su desarrollo, por ejemplo para mejorar las prestaciones, pueden provocar una reestructuración del esquema lógico. De este modo, entre el diseño físico y el diseño lógico hay una realimentación.

En general, el propósito del diseño físico es describir cómo se va a implementar físicamente el esquema lógico obtenido en la fase anterior. Concretamente, en el modelo relacional, esto consiste en:

- Obtener un conjunto de sentencias para crear las tablas de la base de datos y para mantener las restricciones que se deben cumplir sobre ellas.
- Determinar las estructuras de almacenamiento y los métodos de acceso que se van a utilizar para conseguir unas prestaciones óptimas.
- Diseñar el modelo de seguridad del sistema.

En los siguientes apartados se detallan cada una de las etapas que componen la fase del diseño físico.

8.1.1. Traducir el esquema lógico

La primera fase del diseño físico consiste en traducir el esquema lógico a un esquema (físico) que se pueda implementar en el SGBD escogido. Para ello, es necesario conocer toda la funcionalidad que éste ofrece.

Sentencias de creación de las tablas

Las tablas se definen mediante el lenguaje de definición de datos del SGBD. Para ello, se utiliza la información producida durante el diseño lógico: el esquema lógico y toda la documentación asociada (diccionario de datos). El esquema físico consta de un conjunto de tablas y, para cada una de ellas, se especifica:

- El *nombre*. Es conveniente adoptar unas reglas para nombrar las tablas, de manera que aporten información sobre el tipo de contenido. Por ejemplo, a las tablas de referencia se les puede añadir el prefijo o el sufijo `REF`, a las tablas que almacenan información de auditoría ponerles el prefijo/sufijo `AUDIT`, a las tablas que sean de uso para un solo departamento, ponerles como prefijo/sufijo las siglas del mismo, etc.

- La *lista de columnas* con sus nombres. De nuevo resulta conveniente adoptar una serie de reglas para nombrarlas. Algunas reglas habituales son: poner el sufijo PK a las claves primarias (PRIMARY KEY), poner el sufijo FK a las claves ajenas (FOREIGN KEY), usar el nombre de la clave primaria a la que se apunta en el nombre de una clave ajena o el nombre de su tabla, usar el mismo nombre para las columnas que almacenan el mismo tipo de información (por ejemplo, si en varias tablas se guarda una columna con la fecha en que se ha insertado cada fila, usar en todas ellas el mismo nombre para dicha columna), etc. Además, para cada columna se debe especificar:
    - Su *dominio*: tipo de datos, longitud y restricciones de dominio (se especifican con la cláusula CHECK).
    - El *valor por defecto*, que es opcional (DEFAULT).
    - Si admite *nulos* o no (NULL/NOT NULL).
- La *clave primaria* (PRIMARY KEY), las *claves alternativas* (UNIQUE) y las *claves ajenas* (FOREIGN KEY), si las tiene.
- Las *reglas de comportamiento* de las claves ajenas (ON UPDATE, ON DELETE).

A continuación, se muestra un ejemplo de la creación de las tablas FACTURAS y LINEAS_FAC (con las que se trabaja en el capítulo 4) utilizando la especificación de SQL del SGBD libre PostgreSQL.

```
CREATE TABLE facturas (
    codfac NUMERIC(6,0) NOT NULL,
    fecha  DATE         NOT NULL,
    codcli NUMERIC(5,0),
    codven NUMERIC(5,0),
    iva    NUMERIC(2,0),
    dto    NUMERIC(2,0),
    CONSTRAINT cp_facturas PRIMARY KEY (codfac),
    CONSTRAINT ca_fac_cli FOREIGN KEY (codcli)
               REFERENCES clientes(codcli)
               ON UPDATE CASCADE ON DELETE SET NULL,
    CONSTRAINT ca_fac_ven FOREIGN KEY (codven)
               REFERENCES vendedores(codven)
               ON UPDATE CASCADE ON DELETE SET NULL,
    CONSTRAINT ri_dto_fac CHECK (dto BETWEEN 0 AND 50)
);

CREATE TABLE lineas_fac (
    codfac NUMERIC(6,0) NOT NULL,
    linea  NUMERIC(2,0) NOT NULL,
```

```
    cant   NUMERIC(5,0) NOT NULL,
    codart VARCHAR(8) NOT NULL,
    precio NUMERIC(6,2) NOT NULL,
    dto    NUMERIC(2,0),
    CONSTRAINT cp_lineas_fac PRIMARY KEY (codfac, linea),
    CONSTRAINT ca_lin_fac FOREIGN KEY (codfac)
               REFERENCES facturas(codfac)
               ON UPDATE CASCADE ON DELETE CASCADE,
    CONSTRAINT ca_lin_art FOREIGN KEY (codart)
               REFERENCES articulos(codart)
               ON UPDATE CASCADE ON DELETE RESTRICT,
    CONSTRAINT ri_dto_lin CHECK (dto BETWEEN 0 AND 50)
);
```

Mantenimiento de restricciones y reglas de negocio

Las actualizaciones que se realizan sobre las tablas de la base de datos deben observar ciertas restricciones o producir determinadas consecuencias que imponen las reglas de funcionamiento de la empresa. Algunos SGBD proporcionan mecanismos que permiten definir restricciones y reglas, y vigilan su cumplimiento.

Un mecanismo para definir restricciones es la cláusula `CONSTRAINT ... CHECK`. Un ejemplo de ella se puede observar en las sentencias de creación de las tablas `FACTURAS` y `LINEAS_FAC`, sobre la columna `dto`. Otro mecanismo son los *disparadores* (`TRIGGER`), que también se utilizan para establecer reglas de negocio en las que se requiere la realización de alguna acción como consecuencia de algún evento.

Hay algunas reglas que no las pueden manejar todos los SGBD, como por ejemplo «a las 20:30 del último día laborable de cada año archivar los pedidos servidos y borrarlos». Para algunas reglas habrá que escribir programas de aplicación específicos. Por otro lado, hay SGBD que no permiten la definición de reglas, por lo que éstas deberán incluirse en los programas de aplicación.

Todas las reglas que se definan deben estar documentadas. Si hay varias opciones posibles para implementarlas, hay que explicar por qué se ha escogido la opción implementada.

8.1.2. Diseñar la representación física

Uno de los objetivos principales del diseño físico es almacenar los datos de modo eficiente. Para medir la eficiencia hay varios factores que se debe tener en cuenta:

- *Rendimiento de transacciones.* Es el número de transacciones que se quiere procesar en un intervalo de tiempo.
- *Tiempo de respuesta.* Es el tiempo que tarda en ejecutarse una transacción. Desde el punto de vista del usuario, este tiempo debería ser el mínimo posible.
- *Espacio en disco.* Es la cantidad de espacio en disco que hace falta para los ficheros de la base de datos. Normalmente, el diseñador querrá minimizar este espacio.

Lo que suele suceder es que todos estos factores no se pueden satisfacer a la vez. Por ejemplo, para conseguir un tiempo de respuesta mínimo puede ser necesario aumentar la cantidad de datos almacenados, ocupando más espacio en disco. Por lo tanto, el diseñador deberá ir ajustando estos factores para conseguir un equilibrio razonable. El diseño físico inicial no será el definitivo, sino que habrá que ir monitorizándolo para observar sus prestaciones e ir ajustándolo como sea oportuno. Muchos SGBD proporcionan herramientas para monitorizar y afinar el sistema.

Hay algunas estructuras de almacenamiento que son muy eficientes para cargar grandes cantidades de datos en la base de datos, pero no son eficientes para el resto de operaciones, por lo que se puede escoger dicha estructura de almacenamiento para inicializar la base de datos y cambiarla, a continuación, para su posterior operación. Los tipos de organizaciones de ficheros disponibles varían en cada SGBD y algunos sistemas proporcionan más estructuras de almacenamiento que otros. Es muy importante que el diseñador del esquema físico sepa qué estructuras de almacenamiento le proporciona el SGBD y cómo las utiliza.

Para mejorar las prestaciones, el diseñador del esquema físico debe saber cómo interactúan los dispositivos involucrados y cómo esto afecta a las prestaciones:

- *Memoria principal.* Los accesos a memoria principal son mucho más rápidos que los accesos a memoria secundaria (decenas o centenas de miles de veces más rápidos). Generalmente, cuanta más memoria principal se tenga, más rápidas serán las aplicaciones. Si no hay bastante memoria disponible para todos los procesos, el sistema operativo debe transferir páginas a disco para liberar memoria (memoria virtual). Cuando estas páginas se vuelven a necesitar, hay que volver a traerlas desde el disco

(fallos de página). A veces, es necesario llevar procesos enteros a disco (*swapping*) para liberar memoria. El hacer estas transferencias con demasiada frecuencia empeora las prestaciones.

- *CPU.* La CPU controla los recursos del sistema y ejecuta los procesos de usuario. El principal objetivo con este dispositivo es lograr que no haya bloqueos de procesos para conseguirla. Si el sistema operativo, o los procesos de los usuarios, hacen muchas demandas de CPU, ésta se convierte en un cuello de botella. Esto suele ocurrir cuando hay muchas faltas de página o se realiza mucho *swapping*.

- *Entrada/salida a disco.* Los discos tienen una velocidad de entrada/salida. Cuando se requieren datos a una velocidad mayor que ésta, el disco se convierte en un cuello de botella. Dependiendo de cómo se organicen los datos en el disco, se conseguirá reducir la probabilidad de empeorar las prestaciones. Los principios básicos que se deberían seguir para repartir los datos en los discos son los siguientes:

  - Los ficheros del sistema operativo deben estar separados de los ficheros de la base de datos.
  - Los ficheros de datos deben estar separados de los ficheros de índices.
  - Los ficheros con los diarios de operaciones deben estar separados del resto de los ficheros de la base de datos.

- *Red.* La red se convierte en un cuello de botella cuando tiene mucho tráfico y cuando hay muchas colisiones.

Cada uno de estos recursos afecta a los demás, de modo que una mejora en alguno de ellos puede influir en otros.

Analizar las transacciones

Para realizar un buen diseño físico es necesario conocer las consultas y las transacciones que se van a ejecutar sobre la base de datos. Esto incluye tanto información cualitativa, como cuantitativa. Para cada transacción, hay que especificar:

- La frecuencia con que se va a ejecutar.

- Las tablas y los atributos a los que accede la transacción, y el tipo de acceso: consulta, inserción, modificación o eliminación. Por ejemplo, los atributos que se modifican a menudo no son buenos candidatos para construir índices.

- Las restricciones temporales impuestas sobre la transacción. Los atributos utilizados en los predicados de la transacción pueden ser candidatos para construir estructuras de acceso.

Escoger las organizaciones de ficheros

El objetivo de este paso es escoger la organización de ficheros óptima para cada tabla. Por ejemplo, un fichero desordenado es una buena estructura cuando se va a cargar gran cantidad de datos en una tabla al inicializarla, cuando la tabla tiene pocas filas, también cuando en cada acceso se deben obtener todas las filas de la tabla, o cuando la tabla tiene una estructura de acceso adicional, como puede ser un índice.

Por otra parte, los ficheros dispersos (*hashing*) son apropiados cuando se accede a las filas a través de los valores exactos de alguno de sus campos (condición de igualdad en el `WHERE`). Si la condición de búsqueda es distinta de la igualdad (búsqueda por rango, por patrón, etc.), entonces la dispersión no es una buena opción.

Algunos SGBD proporcionan otras organizaciones alternativas a éstas. Las organizaciones de ficheros elegidas deben documentarse, justificando en cada caso la opción escogida.

Escoger los índices a crear y sus tipos

Los índices son estructuras adicionales que se utilizan para acelerar el acceso a las tablas en respuesta a ciertas condiciones de búsqueda. Algunos tipos de índices, los denominados caminos de acceso secundario, no afectan al emplazamiento físico de los datos en el disco y lo que hacen es proporcionar caminos de acceso alternativos para encontrar los datos de modo eficiente basándose en los campos de indexación. Hay que tener en cuenta que los índices conllevan un coste de mantenimiento que es necesario sopesar frente a la ganancia en prestaciones.

Cada SGBD proporcionará uno o varios tipos de índices entre los que escoger. Los más habituales son los índices basados en árboles B+ (o árboles B*) y los basados en la dispersión (*hash*).

Un índice con estructura de árbol B+ es un árbol de búsqueda que siempre está equilibrado (todas las hojas se encuentran al mismo nivel) y en el que el espacio desperdiciado por la eliminación, si lo hay, nunca será excesivo. Los algoritmos para insertar y eliminar son complejos para poder mantener estas restricciones. No obstante, la mayor parte de las inserciones y eliminaciones son procesos simples que se complican sólo en circunstancias especiales: cuando se intenta insertar en un nodo que está lleno o cuando se intenta borrar en un nodo que está ocupado hasta la mitad. Las simulaciones muestran que un índice con estructura de árbol B+ de cuatro niveles contiene unos cien millones de nodos hoja, lo que indica que en cuatro accesos se puede llegar a los datos, incluso si la tabla es muy grande. Este tipo de índices es útil tanto en búsquedas con la condición de igualdad sobre el campo de indexación, como para hacer búsquedas por rangos.

Un índice basado en la dispersión es un fichero disperso en el que las entradas se insertan en el índice aplicando una función sobre el campo de indexación.

Aunque el acceso a los datos es muy rápido (es casi un acceso directo), este tipo de índices sólo pueden usarse cuando la condición de búsqueda es la igualdad sobre el campo de indexación.

A la hora de seleccionar los índices a crear, se pueden seguir las siguientes indicaciones:

- Crear un índice sobre la clave primaria de cada tabla.

  La mayor parte de los SGBD relacionales crean un índice único de manera automática sobre la clave primaria de cada tabla porque es el mecanismo que utilizan para mantener la unicidad.

- No crear índices sobre tablas pequeñas. Si el SGBD ha creado índices automáticamente sobre este tipo de tablas, se pueden eliminar (`DROP INDEX`).

  Aquí conviene tener en cuenta que, en la mayor parte de los SGBD, no se permite eliminar un índice creado sobre una clave primaria a la que apunta una clave ajena, ya que este índice se utiliza para mantener la integridad referencial.

- Crear un índice sobre las claves ajenas que se utilicen con frecuencia en operaciones de `JOIN`.

- Crear un índice sobre los atributos que se utilizan con frecuencia para hacer restricciones `WHERE` (son condiciones de búsqueda).

- Crear un índice único sobre las claves alternativas que se utilizan para hacer búsquedas.

  Al igual que ocurre con las claves primarias, los SGBD suelen mantener la unicidad de las claves alternativas mediante un índice único que crean automáticamente.

- Evitar los índices sobre atributos que se modifican a menudo.

- Evitar los índices sobre atributos poco selectivos: aquellos en los que la consulta selecciona una porción significativa de la tabla (más del 15 % de las filas).

- Evitar los índices sobre atributos formados por tiras de caracteres largas.

- Evitar los índices sobre tablas que se actualizan mucho y que se consultan muy esporádicamente (tablas de auditoría o diarios). Si se han creado índices sobre este tipo de tablas, podría ser aconsejable eliminarlos.

- Revisar si hay índices redundantes o que se solapan y eliminar los que no sean necesarios.

Estimar la necesidad de espacio en disco

El diseñador debe estimar el espacio necesario en disco para la base de datos. Esto es especialmente importante en caso de que se tenga que adquirir nuevo equipamiento informático. Esta estimación depende del SGBD que se vaya a utilizar y del *hardware*. En general, se debe estimar el número de filas de cada tabla y su tamaño. También se debe estimar el factor de crecimiento de cada tabla.

8.1.3. Diseñar los mecanismos de seguridad

Los datos constituyen un recurso esencial para la empresa, por lo tanto su seguridad es de vital importancia. Durante el diseño lógico se habrán especificado los requerimientos en cuanto a seguridad que en esta fase se deben implementar. Para llevar a cabo esta implementación, el diseñador debe conocer las posibilidades que ofrece el SGBD que se vaya a utilizar.

Diseñar las vistas de los usuarios

El objetivo de este paso es diseñar las vistas o esquemas externos de los usuarios, correspondientes a los esquemas lógicos de cada grupo de usuarios. Cada esquema externo estará formado por tablas y vistas (`VIEW`) de SQL. Las vistas, además de preservar la seguridad, mejoran la independencia de datos, reducen la complejidad y permiten que los usuarios vean los datos en el formato deseado.

Diseñar las reglas de acceso

El administrador de la base de datos asigna a cada usuario un identificador que tendrá una contraseña asociada por motivos de seguridad. Para cada usuario o grupo de usuarios se otorgarán privilegios para realizar determinadas acciones sobre determinados objetos de la base de datos. Por ejemplo, los usuarios de un determinado grupo pueden tener permiso para consultar los datos de una tabla concreta, y no tener permiso para actualizarlos.

8.1.4. Monitorizar y afinar el sistema

Una vez implementado el esquema físico de la base de datos, ésta se debe poner en marcha para observar sus prestaciones. Si éstas no son las deseadas, el esquema deberá cambiar para intentar satisfacerlas. Una vez afinado el esquema, éste no permanecerá estático, ya que tendrá que ir cambiando conforme lo requieran los nuevos requisitos de los usuarios. Los SGBD proporcionan herramientas para monitorizar el sistema mientras está en funcionamiento.

8.2. Vistas

Hay tres características importantes inherentes a los sistemas de bases de datos: la separación entre los programas de aplicación y los datos, el manejo de múltiples vistas por parte de los usuarios (esquemas externos) y el uso de un catálogo o diccionario para almacenar el esquema de la base de datos. En 1975, el comité ANSI-SPARC (*American National Standard Institute- Standards Planning and Requirements Committee*) propuso una arquitectura de tres niveles para los sistemas de bases de datos, que resulta muy útil a la hora de conseguir estas tres características.

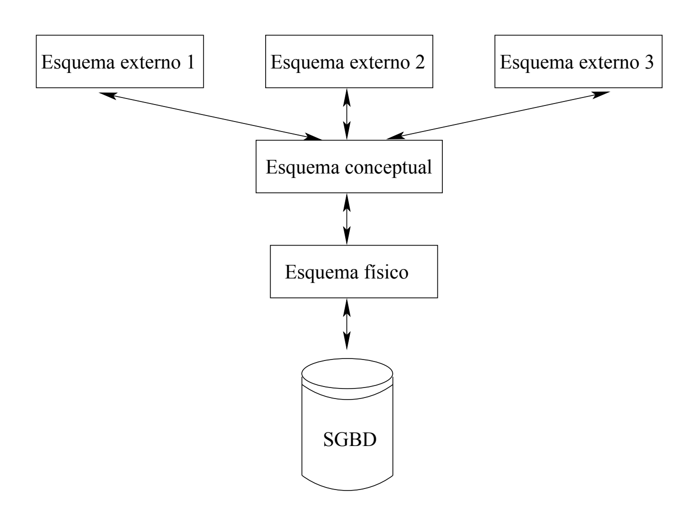


Figura 8.1: Arquitectura ANSI-SPARC para los Sistemas de Bases de Datos.

El objetivo de la arquitectura de tres niveles es el de separar los programas de aplicación de la base de datos física. En esta arquitectura, el esquema de una base de datos se define en tres niveles de abstracción distintos (ver figura 8.1):

1. En el *nivel interno* se describe la estructura física de la base de datos mediante un *esquema interno*. Este esquema se especifica mediante un modelo físico y describe todos los detalles para el almacenamiento de la base de datos, así como los métodos de acceso.

2. En el *nivel conceptual* se describe la estructura de toda la base de datos para una comunidad de usuarios (todos los de una empresa u organización), mediante un *esquema conceptual*. Este esquema oculta los detalles de las estructuras de almacenamiento y se concentra en describir entidades, atributos, relaciones, operaciones de los usuarios y restricciones. En este nivel se puede utilizar un modelo conceptual o un modelo lógico para especificar el esquema.

3. En el *nivel externo* se describen varios *esquemas externos* o *vistas de usuario*. Cada esquema externo describe la parte de la base de datos que interesa a un grupo de usuarios determinado y oculta a ese grupo el resto de la base de datos. En este nivel se puede utilizar un modelo conceptual o un modelo lógico para especificar los esquemas.

La mayoría de los SGBD no distinguen del todo los tres niveles. Algunos incluyen detalles del nivel físico en el esquema conceptual. En casi todos los SGBD que se manejan vistas de usuario, los esquemas externos se especifican con el mismo modelo de datos que describe la información a nivel conceptual, aunque en algunos se pueden utilizar diferentes modelos de datos en los niveles conceptual y externo.

Hay que destacar que los tres esquemas no son más que descripciones de los mismos datos pero con distintos niveles de abstracción. Los únicos datos que existen realmente están a nivel físico, almacenados en un dispositivo como puede ser un disco. En un SGBD basado en la arquitectura de tres niveles, cada grupo de usuarios hace referencia exclusivamente a su propio esquema externo.

La arquitectura de tres niveles es útil para explicar el concepto de *independencia de datos*, que se puede definir como la capacidad para modificar el esquema en un nivel del sistema sin tener que modificar el esquema del nivel inmediato superior. Se pueden definir dos tipos de independencia de datos:

- La *independencia lógica* es la capacidad de modificar el esquema conceptual sin tener que alterar los esquemas externos ni los programas de aplicación. Se puede modificar el esquema conceptual para ampliar la base de datos o para reducirla. Si, por ejemplo, se reduce la base de datos eliminando una entidad, los esquemas externos que no se refieran a ella no deberán verse afectados.

- La *independencia física* es la capacidad de modificar el esquema interno sin tener que alterar el esquema conceptual (o los externos). Por ejemplo, puede ser necesario reorganizar ciertos ficheros físicos con el fin de mejorar el rendimiento de las operaciones de consulta o de actualización de datos. Dado que la independencia física se refiere sólo a la separación entre las aplicaciones y las estructuras físicas de almacenamiento, es más fácil de conseguir que la independencia lógica.

Cada esquema externo estará formado por un conjunto de tablas (TABLE) y un conjunto de vistas (VIEW). En la arquitectura de tres niveles estudiada, se describe una vista externa como la estructura de la base de datos tal y como la ve un usuario en particular. En el modelo relacional, el término vista tiene un significado un tanto diferente. En lugar de ser todo el esquema externo de un usuario, una vista es una *tabla virtual*, una tabla que en realidad no existe como tal.

Una vista es el resultado dinámico de una o varias operaciones relacionales realizadas sobre las tablas. La vista es una tabla virtual que se produce cuando un usuario la consulta. Al usuario le parece que la vista es una tabla que existe y la puede manipular como si se tratara de una tabla, pero la vista no está almacenada físicamente. El contenido de una vista está definido como una consulta sobre una o varias tablas.

En SQL, la sentencia que permite definir una vista es la siguiente:

```
CREATE VIEW nombre_vista [ ( nombre_col, ... ) ]
      AS sentencia_SELECT
      [ WITH CHECK OPTION ];
```

Las columnas de la vista se pueden nombrar especificando la lista entre paréntesis. Si no se especifican nuevos nombres, los nombres son los mismos que los de las columnas de las tablas especificadas en la sentencia SELECT.

La opción WITH CHECK OPTION impide que se realicen inserciones y actualizaciones sobre la vista que no cumplan las restricciones especificadas en la misma. Por ejemplo, si se crea una vista que selecciona los clientes con códigos postales de la provincia de Castellón (aquellos que empiezan por 12) y se especifica esta cláusula, el sistema no permitirá actualizaciones de códigos postales de clientes de esta provicia si los nuevos códigos postales son de una provincia diferente. Del mismo modo, a través de la vista sólo será posible insertar clientes con códigos postales de Castellón. Es como si se hubiera establecido una restricción de tipo CHECK con el predicado del WHERE de la definición de la vista.

Cualquier operación que se realice sobre la vista se traduce automáticamente a operaciones sobre las tablas de las que se deriva. Las vistas son dinámicas porque los cambios que se realizan sobre las tablas que afectan a una vista se reflejan inmediatamente sobre ella. Cuando un usuario realiza un cambio sobre la vista (no todo tipo de cambios están permitidos), este cambio se realiza sobre las tablas de las que se deriva.

Las vistas son útiles por varias razones:

- Proporcionan un poderoso mecanismo de seguridad, ocultando partes de la base de datos a ciertos usuarios. El usuario no sabrá que existen aquellos atributos que se han omitido al definir una vista.

- Permiten que los usuarios accedan a los datos en el formato que ellos desean o necesitan, de modo que los mismos datos pueden ser vistos con formatos distintos por distintos usuarios.

```
CREATE VIEW domicilios (codcli,nombre,direccion,poblacion) AS
   SELECT c.codcli,c.nombre,c.direccion,c.codpostal || ' - '
          || pu.nombre || ' (' || pr.nombre || ')'
   FROM clientes c JOIN pueblos pu USING(codpue)
                   JOIN provincias pr USING(codpro);

SELECT * FROM domicilios;

codcli nombre direccion poblacion
------ ------ --------- ------------------------------
210    Luis   C/Pez, 3  12540 - Vila-real (Castellón)
```

- Se pueden simplificar operaciones sobre las tablas que son complejas. Por ejemplo, se puede definir una vista que muestre cada vendedor con el nombre de su jefe:

```
CREATE VIEW vj ( codven, nombreven, codjefe, nombrejefe )
   SELECT v.codven, v.nombre, j.codven, j.nombre
   FROM   vendedores v LEFT OUTER JOIN vendedores j
          ON (v.codjefe=j.codven);
```

  El usuario puede hacer restricciones y proyecciones sobre la vista, que el SGBD traducirá en las operaciones equivalentes sobre el `JOIN`.

```
SELECT f.codfac, f.fecha, vj.vendedor, vj.jefe
FROM   facturas f JOIN vj USING(codven)
WHERE  ... ;
```

- Las vistas proporcionan independencia de datos a nivel lógico, que también se da cuando se reorganiza el nivel conceptual. Si se añade un atributo a una tabla, los usuarios no se percatan de su existencia si sus vistas no lo incluyen. Si una tabla existente se reorganiza o se divide en varias tablas, se pueden crear vistas para que los usuarios la sigan viendo como al principio.

- Las vistas permiten que se disponga de información expresada en forma de reglas generales de conocimiento relativas al funcionamiento de la organización. Una de estas reglas puede ser «los artículos en oferta son los que tienen descuento» y se puede definir una vista que contenga sólo estos artículos, aunque ninguna columna de la base de datos indique cómo ha de considerarse cada artículo (es el conocimiento).

```
CREATE VIEW articulos_oferta AS
   SELECT *
   FROM   articulos
   WHERE  dto > 0;
```

Cuando se actualiza una tabla, el cambio se refleja automáticamente en todas las vistas que la referencian. Del mismo modo, si se actualiza una vista, las tablas de las que se deriva deberían reflejar el cambio. Sin embargo, hay algunas restricciones respecto a los tipos de modificaciones que se pueden realizar sobre las vistas. En el estándar de SQL se definen las condiciones bajo las que una vista es actualizable o es insertable. Básicamente, una vista es actualizable si se puede identificar de modo único la fila a la que afecta la actualización.

- Una vista definida sobre varias tablas es actualizable si contiene las claves primarias de todas ellas y los atributos que no aceptan nulos.
- Una columna de una vista definida sobre varias tablas se podrá actualizar si se obtiene directamente de una sola de las columnas de alguna de las tablas y si la clave primaria de dicha tabla está incluida en la vista.
- Las vistas definidas con operaciones de conjuntos pueden ser actualizables, pero no son insertables (no se puede determinar en qué tabla hacer la inserción).

Ya que el estándar permite que sean actualizables un conjunto muy restringido de vistas, en ocasiones será necesario hacer que una vista sea actualizable mediante disparadores o reglas del tipo *en lugar de*.

Bibliografía

[1] C. Batini, S. Ceri, S. B. Navathe (1994)
Diseño Conceptual de Bases de Datos. Un enfoque de entidades–interrelaciones. Addison–Wesley / Díaz de Santos.

[2] M. Celma, J. C. Casamayor, L. Mota (2003)
Bases de Datos Relacionales. Pearson – Prentice Hall.

[3] T. Connolly, C. Begg, A. Strachan (1998)
Database Systems. A Practical Approach to Design, Implementation and Management. Segunda edición. Addison–Wesley.

[4] C. J. Date (1995)
An Introduction to Database Systems. Sexta Edición. Addison–Wesley.

[5] R. Elmasri, S. B. Navathe (2002)
Fundamentos de Sistemas de Bases de Datos.
Tercera Edición. Addison–Wesley.

[6] M. J. Hernández (1997)
Database Design for Mere Mortals.
Addison–Wesley Developers Press.

[7] R. Ramakrishnan, J. Gehrke (2003)
Database Management Systems.
Tercera Edición. McGraw–Hill.